(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PETRI NET MODELING AND AUTOMATED SOFTWARE SAFETY ANALYSIS: METHODOLOGY FOR AN EMBEDDED MILITARY APPLICATION

by

Alan D. Lewis

June 1988

Thesis Advisor: Daniel L. Davis

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified			1b Restrictive Markings
2a Security Classification Authority			3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification Downgrading Schedule			
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (*If applicable*): 33		7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000			7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding Sponsoring Organization	8b Office Symbol (*If applicable*)		9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)			10 Source of Funding Numbers
		Program Element No	Project No / Task No / Work Unit Accession No

11 Title (*Include security classification*): PETRI NET MODELING AND SOFTWARE SAFETY ANALYSIS: METHODOLOGY FOR AN EMBEDDED MILITARY APPLICATION

12 Personal Author(s): Alan D. Lewis

13a Type of Report	13b Time Covered	14 Date of Report (*year, month, day*)	15 Page Count
Master's Thesis	From To	June 1988	98

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	Petri nets, software safety, missile fuze, safety arming device, Petri Net Utilities, P-NUT

19 Abstract (*continue on reverse if necessary and identify by block number*)

This thesis investigates the feasibility of software safety analysis using Petri net modeling and an automated suite of Petri Net UTilities (P-NUT) developed at UC Irvine. We briefly introduce software safety concepts, Petri nets, reachability, and the use of P-NUT. We then develop a methodology to combine these ideas for efficient and effective preliminary safety analysis of a real-time, embedded software, military system.

20 Distribution Availability of Abstract	21 Abstract Security Classification	
☒ unclassified unlimited ☐ same as report ☐ DTIC users	Unclassified	
22a Name of Responsible Individual: Daniel L. Davis	22b Telephone (*Include Area code*): (408) 646-3390	22c Office Symbol: 52Dv

DD FORM 1473,84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

Petri Net Modeling and Software Safety Analysis: Methodology for an Embedded Military Application

by

Alan D. Lewis
Lieutenant, United States Navy
B.S., United States Naval Academy, 1980

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

Author: Alan D. Lewis

Approved by:

Daniel L. Davis, Thesis Advisor

Uno R. Kodres, Second Reader

Vincent Y. Lum, Chairman, Department of Computer Science

Gordon E. Schacher, Dean of Science and Engineering

ABSTRACT

This thesis investigates the feasibility of software safety analysis using Petri net modeling and an automated suite of Petri Net UTilities (P-NUT) developed at UC Irvine. We briefly introduce software safety concepts, Petri nets, reachability theory, and the use of P-NUT. We then develop a methodology to combine these ideas for efficient and effective preliminary safety analysis of a real-time, embedded software, military system.

TABLE OF CONTENTS

I. INTRODUCTION

> Computers are increasingly being used as passive (monitoring) and active (controlling) components of real-time systems, e.g., air traffic control, aerospace, aircraft, industrial plants, and hospital patient monitoring systems. The problems of safety become important when these applications include systems where the consequences of failure are serious and may involve grave danger to human life and property. [Leveson and Stolzy, 1987]

Unfortunately, little is known about applying safety considerations to the design and evaluation of computer-controlled real-time systems. The military relies heavily on safety-critical, computer-controlled, real-time systems and has published several standards for test and verification of software system safety (MIL-STD-SNS, MIL-STD-882B, MIL-STD-1574A). MIL-STD-882B(DoD) contains requirements for software hazard analysis and software safety verification, while MIL-STD-1574A(USAF) lists the requirements for software safety analysis and integrated system (hardware, software, and interfaces) safety. MIL-STD-SNS(USN) covers software safety analysis for nuclear weapons systems.

Problems with ascertaining and verifying the safety of a software-controlled system include the difficulty of providing realistic test conditions and simulating hardware errors, transient faults, and system interfaces. There is no existing language which incorporates the myriad system facets, such as software, hardware, and the resulting

interfaces. This overall system view is critical, as the greatest source of problems encountered in computer controlled systems may be the lack of system level methods. [Leveson, 1986]

There are several proposed techniques for software safety analysis, including Petri net modeling [Leveson and Stolzy, 1987], Fault Tree Analysis [Vesely et al., 1981], and Real-Time Logic (RTL) [Jahanian and Mock, 1986]. This thesis follows the Leveson and Stolzy use of Petri Net modeling and the other techniques will not be discussed further. For a brief synopsis on the other methods, see Hayward [1987].

The system under investigation is a proposed air-to-air guided missile safety and arming device, developed at the Naval Weapons Center in China Lake, California. Although this particular safety arming device was never actually constructed, a software prototype was written and tested. This device is excellent for developing a methodology to analyze safety-critical computer/software-controlled systems. The device is nontrivial, contains embedded software, and if designed incorrectly or tested ineffectively might result in personal injury or unwanted property destruction.

This thesis refines and continues the work of Duston Hayward [1987], who initially investigated the practical feasibility of using Petri nets, and the Leveson and Stolzy [1987] analysis techniques to meet military standards.

Beginning with the safety and arming device software assembler code, and verbal descriptions of the components, Hayward [1987] constructed system and system software flowcharts and Petri net models of system components. He then combined the flowcharts into a single system description by conversion to a Petri net model. Using the safety and arming device, Hayward demonstrated techniques for manual construction of partial reachability graphs and application of Leveson and Stolzy [1987] safety analysis methods.

Following publication of Hayward [1987], the U. S. Naval Postgraduate School received a set of automated Petri net analysis and utility tools from the Department of Information and Computer Science, University of California, Irvine [Morgan and Razouk, 1985; Razouk, 1987; Morgan, 1987]. These utilities construct the reachability graph of an entered net and support automated reachability analysis through use of a sophisticated reachability graph analyzer. Our work is the first known application of these automated utilities to the area of software safety analysis.

We begin with brief introductions to software safety, Petri nets, reachability theory, and use of the Petri Net UTilities (P-NUT). We discuss refinements made to the Hayward [1987] model and develop a methodology for efficient and effective preliminary safety analysis of a complex, safety-critical, software-controlled system.

II. INTRODUCTION TO SOFTWARE SAFETY

A. WHAT IS SOFTWARE SAFETY?

The American College Dictionary defines safety as the quality of insuring against hurt, injury, danger, or risk. It follows that software safety may be considered as freedom from software causing danger or risk. Software, however, is inherently safe, since alone it can do no physical damage. Although it is the hardware that the software controls which actually presents the hazard, we must treat software and hardware as one entity for analysis purposes. "Software engineering techniques that do not consider the system as a whole, including the interactions between the hardware, software, and human operators, will have limited usefulness for real-time control software." [Leveson, 1986] The safety of a software-controlled system is commonly referred to as software safety.

Safety should not be confused with reliability. Safety is the probability that a mishap (accident) will not occur regardless of whether or not the intended function is performed. Reliability is normally defined, in the engineering community, as the probability that the system will accomplish its intended function for a specified time under specified environmental conditions [Ericson, 1981; Konakovsky, 1978; Leveson, 1986]. These are quite different

concepts, as demonstrated in the analysis of munitions. One would expect that when the reliability of a weapon is improved, the weapon becomes less safe. Improvements that increase the probability of detonation may very well increase the likelihood of accidental firing, unless specific precautions are made in the design to improve the safety as reliability is improved. [Roland and Moriarity, 1983; Leveson, 1986]

B. INTRODUCTION TO SOFTWARE SAFETY ANALYSIS

To ensure system safety, it is necessary to show that the software and hardware will perform as required and verify that the relationships between software, hardware, and system behavior are correct.

> Many of the system safety techniques that have been developed to aid in building electromechanical systems with minimal risk do not seem to apply when computers are introduced. The major differences appear to stem from the lack of system-level approaches to building software controlled systems. [Leveson 1986]

Current system safety techniques do not consider human design errors in system failures. Human errors are assumed never to have occurred or to have been removed prior to delivery and operation. With the growth of embedded software systems and powerful microprocessors, the complexity of software and hardware has grown tremendously and resulted in a nonlinear increase in human error design flaws. Due to

system complexity, it may be impossible to prove correctness and safety of a realistic control system. [Lauber, 1980]

Based on this situation, Leveson [1986] argued the need for a new approach to the software safety analysis problem. The "black box" approach to software is not valid. A total system concept must be employed to properly account for software effects on the system.

The initial stage of the analysis is to focus on system failures that have the most drastic consequences. This is especially useful in situations where the system under investigation has relatively few failures leading to mishaps. The technique is to start with a given set of unacceptable failures and then by means of a "backward" approach ensure that the failures are eliminated, or at least minimized. [Leveson, 1986]

One method for combining software, hardware, human operators, and system interfaces is by timed and untimed Petri net modeling [Leveson and Stolzy, 1987]. The Petri net model successfully treats all aspects of the system as integral parts of the whole.

This thesis will follow Hayward's [1987] work, which employed Leveson's untimed Petri net approach to system modeling. We will focus on the initial stages of the safety analysis, e.g., potential "catastrophic" failure determination and evaluation. Methodologies will be

presented for untimed Petri net modeling of nontrivial system components and for using available automated techniques in preliminary safety analysis.

III. PETRI NETS AND REACHABILITY

A. INTRODUCTION TO PETRI NETS

Petri nets were originally developed by A. W. Holt and others, based on the theories of Carl Adam Petri [Petri, 1962]. Petri's efforts were directed to presenting a theory for the asynchronous flow of communication between computer components. Holt demonstrated that Petri nets could be used to effectively model concurrent systems because they have the ability to model parallelism and synchronization. This thesis will assume the reader has little familiarity with Petri nets. An excellent source for more information can be found in Peterson [1981].

In computer science terminology, Petri nets are directed graphs whose nodes are transitions and places. Places model conditions, and transitions model the occurrence of events. The firing of a transition is considered to be instantaneous, therefore no two events can happen simultaneously. Inputs to a transition represent the preconditions of the event, while outputs of the transition are the postconditions. In Figure 3-1, the arcs of the graph (denoted by arrows) denote those places (denoted by circles) which are inputs to the transitions (denoted by bars) and those which are outputs. Each place contains zero or more tokens, which represent the holding of a condition. The number of tokens contained in a

place is called the marking of that place [Peterson, 1981]. The marking or "state" of the entire net consists of the set of markings of all individual places within the net.

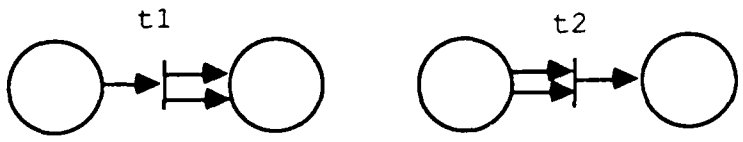


Figure 3-1. **Basic Petri Net Structures**

Figure 3-1 shows two basic arrangements of Petri net primitives. In Figure 3-1, arrows (arcs) coming out of the circles (precondition places) and terminating into the vertical bars (transitions) represent the number of input tokens required to enable the transitions. The number of arcs coming out of the transitions signifies the number of tokens that will be created when the transitions occur. These newly created tokens will be deposited into the circles (postcondition places) where the arrows terminate. Note that there is no dependency between the number of input arcs required to enable a given transition and that transition's number of output arcs. When a Petri net transition fires, the enabling tokens are consumed and the output tokens are created.

A transition is "enabled" when there is a minimum of one token on each of its input arcs. Figure 3-2 shows the structures of Figure 3-1 with tokens in the input places.

These are examples of enabled transitions because there is one token for each transition input arc. If there are less input tokens than there are input arcs to the transition, the transition is not enabled and cannot fire.

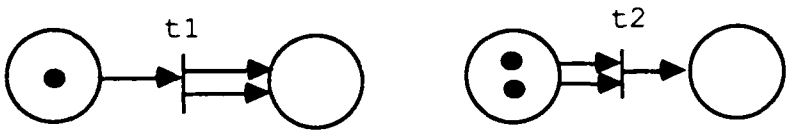


Figure 3-2. **Examples of Enabled Transitions**

Figure 3-3 depicts the basic structures from Figure 3-2 after the transitions have fired. When a transition "fires," one token is placed in the output place(s) for each output arc. Notice that the input tokens have been consumed and output tokens created.

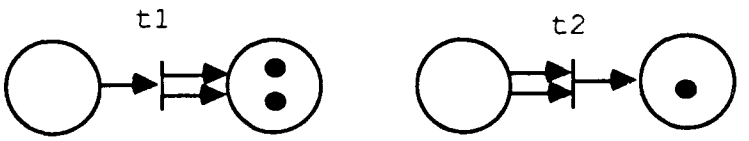


Figure 3-3. **Basic Petri Net Structures After Transitions Firing**

It is important to understand nondeterminism as it relates to untimed Petri nets. An enabled transition may fire instantly, at some later time, or never. In a situation such as Figure 3-4, either t1 or t2 may fire, and either transition has equal probability of occurring or never

occurring. Furthermore, if more than one transition in a given net is enabled, then any of the enabled transitions may be the next to fire. It is this feature that makes Petri nets particularly suitable for concurrent system modeling [Peterson, 1981].

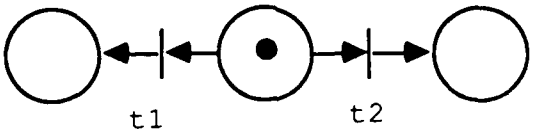


Figure 3-4. **Either t1 or t2 May Fire**

Timed Petri nets remove much of the nondeterministic nature of the net by adding minimum and maximum allowable transition firing times. In the scope of our work in control and information flow modeling, only untimed nets were used. Modeling and automated safety analysis of timed nets is left for future research.

Since Petri nets are used to model events and activities in a given system, they are particularly suited to model flow of information or control.

B. PETRI NET THEORY

The formal definition of Petri nets, using the notation of Peterson [1981], follows. A Petri net is composed of a set of places P, a set of transitions T, an input function I, an output function O, and an initial marking, μ_0.

Definition: Petri net structure, Φ, is a 5-tuple $C=(P,T,I,O,\mu_0)$.

$P = \{p_1,p_2,\ldots,p_n\}$ is a finite set of places, $n \geq 0$.

$T = \{t_1,t_2,..,t_m\}$ is a finite set of transitions, $m \geq 0$. The set of places and the set of transitions are disjoint, $P \cap T$ = null set, $\emptyset$.

$I : T \rightarrow P^{\infty}$ is the input function, a mapping from transitions to bags of places.

$O : T \rightarrow P^{\infty}$ is the output function, a mapping from transitions to bags of places.

$\mu_0 : P \rightarrow N$ is the initial marking for the net where N is the set of nonnegative integers.

Definition: A transition t_j can fire if and only if it is enabled.An enabled transition may fire at any time or may never fire.

Definition: The multiplicity of an input place p_i for a transition t_J is the number of occurrences of the place in the input bag of the transition, $\#(p_i, I(t_J))$.

Definition: transition t_J is an input of place p_i if p_i is an output of t_J.

$$\#(t_J, I(p_i)) = \#(p_i, O(t_J))$$

$I : P \rightarrow T^{\infty}$

$O : P \rightarrow T^{\infty}$

transition t_J is an output of place p_i if p_i is an input of t_J.

$$\#(t_J, O(p_i)) = \#(p_i, I(t_J))$$

Definition: The state of the net, σ, consists of the marking of all places within the net.

C. REACHABILITY

The state of a system is defined by the set of conditions or markings that exist within the Petri net representation of the system at any given instant. Consequently, the state of a system is always well defined by the the set of states of individual places within the system.

In fundamental terms, reachability is the possibility that a given initial condition (state) could lead to a given final condition (state). If there is any possible state sequence from the initial state to the specified state, the specified state is said to be reachable from the initial state. A Petri net reachability graph is a directed graphical depiction of all possible state sequences beginning with the initial state of the net. In a reachability graph, nodes represent states and the root node is the initial state. Arcs between state nodes represent sets of transitions which, if fired, would take the net sequentially from one state to another. State reachability analysis is solely concerned with the possibility of any sequence of states (graph nodes) and transition firings (graph arcs) taking the system from a given initial state to a specified final state.

Petri net safety analysis uses reachability to determine the possibility of mishap states. A Petri net reachability state set is the set of all states in the net reachability graph. This set can be further divided into two disjoint

subsets. One subset of states has the possibility of reaching either high- or low-risk states, while the other subset can reach only low-risk states. A critical state is a low-risk state which can either lead to a set of high-risk states or to a set of other low-risk states. If the final critical state on a path leading to a set of high-risk states follows the high-risk path, there is no further possibility of returning to the low-risk state set. If a reachable high-risk state exists, there must be a critical state somewhere on the state sequence path from the initial state to the high-risk state. [Leveson, 1986]

One approach for the elimination of paths terminating in high-risk states is proposed by Leveson [1986]. This method begins with high-risk state determination and works backward along the state sequence to identify the first critical state encountered. Design techniques are then used to ensure that the high-risk path is never taken. This approach is appropriately named Backward Reachability Analysis.

Our safety analysis work was primarily concerned with identifying mishap states of the system and determining their reachability from the initial state. For a formal description of reachability theory see Peterson [1981] or Leveson [1986].

IV. PETRI NET UTILITIES (P-NUT)

A. INTRODUCTION TO P-NUT

The Petri Net UTilities (P-NUT) were developed by the computer science department of the University of California, Irvine. The tools were constructed to assist researchers in applying Petri net analysis techniques to the design of complex concurrent systems. Our work employed P-NUT Version 2.2 installed on a SUN 3 computer with an enhanced UNIX 4.2BSD operating system. User manuals [Razouk, 1987; Morgan, 1987] contain necessary installation information and provide guidelines for the translation of graphical Petri nets to P-NUT compatible input text files.

P-NUT creates and manipulates three usable object types: Petri nets, reachability graphs, and execution traces [Razouk, 1987]. Our work did not use execution traces and they will not be discussed further.

Petri nets are input to P-NUT in a text format and transformed to an internal representation using the translation (**transl**) tool. The Reachability Graph Builder (RGB) uses the translated file to build a reachability graph, which can then be analyzed by the Reachability Graph Analyzer (RGA).

Our work consisted of creating an untimed Petri net text file, translating the file to RGA internal representation

form, constructing the reachability graph, and analyzing the reachable states. We will present only the necessary methodology, from Razouk [1987] and Morgan [1987], to accomplish these tasks.

B. TRANSLATING THE PETRI NET

The first step in creating a Petri net on P-NUT is to provide a textual version of the graphical net. The translator tool (**transl**) is then used to transform this textual net to suitable internal RGA format. Any text editor may be used for initial text file creation. The textual file consists of a net transition listing. Each transition's inputs (precondition places) and outputs (postcondition places) are specified, one transition per line. To promote effective analysis, we highly recommend numbered transitions and meaningful place names.

The text listing of each transition must begin with the transition number (or name) enclosed by colons, i.e., :t0:. The transition number is followed by a comma-separated list of input places required to enable the transition. If more than one token is required in any input place, the number is specified by enclosing it in parentheses following the input place name. Following the last input place of a transition, the symbol "->" signifies that the output places follow. Output places are listed in the same manner as input places. As with input places, if more than one token will be gained

by an output place after the transition fires, the number must be specified in parentheses following that output place.

Following a listing of all transitions, the initial conditions, or marking, of the net must be specified. The initial conditions consist of a comma-separated list of places that contain tokens and are enclosed by "< >." If any place initially contains more than one token, that number must be specified in parentheses as described above. Comments are allowed but must be indicated by enclosure within "/* */" . If any transition requires more than one line, the use of a reverse diagonal "\" followed by a carriage return is interpreted by P-NUT as a space character. An example of a simple Petri net is given in Figure 4-1.

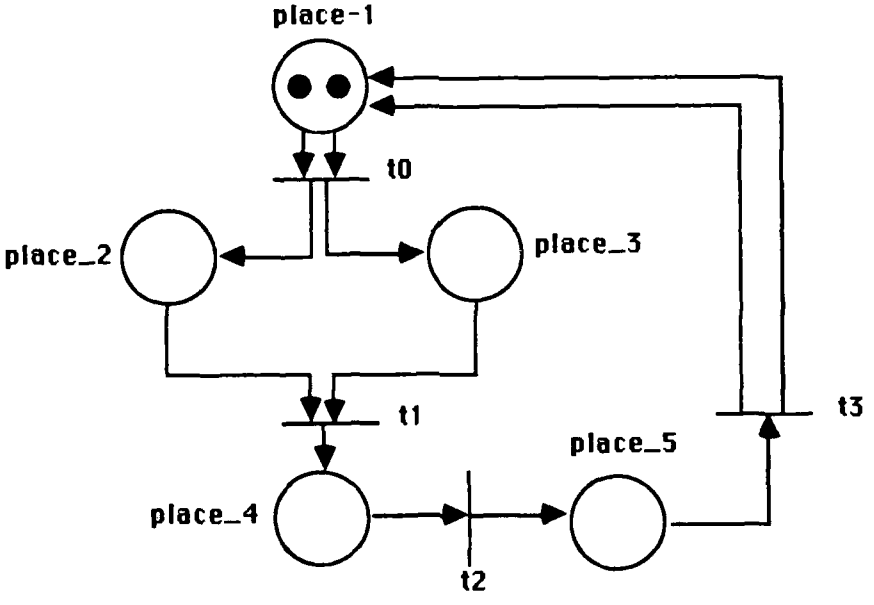


Figure 4-1. **A Simple Petri Net**

The P-NUT textual input file version of the Figure 4-1 Petri net is contained in Figure 4-2.

```
:t0: place_1(2) -> place_2,place_3
:t1: place_2,place_3 -> place_4
:t2: place_4 -> place_5
:t3: place_5 -> place_1(2)
<place_1(2)>   /* initial conditions */
```

Figure 4-2. **P-NUT Text Version of Figure 4-1 Petri Net**

The net in Figure 4-1 contains four transitions and five places. The initial transition is numbered zero, reflecting the internal transition numbering sequence used within P-NUT. Note that transition t0 is not enabled unless two tokens are contained in place_1, and that when transition t3 fires, two tokens will be gained by place_1. The initial conditions are two tokens in place_1.

Assume this file is named "example_1." **pnl** (P-NUT lint) is a tool that scans the initial text file for syntactic and semantic errors prior to translation. The command to invoke **pnl** for example 1 is **pnl example_1**. The output of this tool will be a short error description. To translate the text file and redirect output to another file (rather than to the terminal) the command is **transl example_1 > example_1.pn**. If no input text file is specified, **transl**

will expect terminal entry. **transl** does not tolerate input errors, therefore exclusive use of input text files is recommended. The choice of output file name is at the discretion of the user, but we recommend the ".pn" suffix to denote that the file is in correct internal P-NUT Petri net format. Any P-NUT tool output can be redirected using the above method.

C. BUILDING AND PRINTING REACHABILITY GRAPHS

Reachability graph nodes represent states and the edges represent possible state transitions. The Reachability Graph Builder (RGB) takes a translated net text file as input and creates an internal representation of the Petri net reachability graph. In the untimed graph, the state of the system is completely described by the token distribution on places. Arcs in the reachability graph denote the path between source and destination states in the system. The basic command to build the reachability graph of translated file is **rgb example_1.pn > example_1.rg**. The ".rg" suffix is recommended. If the Petri net is known to always have less than 127 tokens possible in any given place, it is called "bounded" at 127, and the command, **rgb -b example_1.pn > example_1.rg** should be used. This option saves both memory and processor time. If the net is known to be bounded at 1, it is called "safe" (not to be confused with low risk), and the command **rgb -s example_1.pn >**

example_1.rg will save even more CPU time and memory. Use of a file for redirected output is recommended for all commands, otherwise output will default to the standard output device.

After building the reachability graph, it can be printed and viewed using the Reachability Graph Printer (RGP). The command **rgp example_1.rg > example_1.g** will print our example graph and redirect output. The ".g" suffix is recommended for informational purposes and to differentiate the file from the internal format of the reachability graph. The important consideration in choosing suffix names for any P-NUT output file is uniformity.

RGP output is a schematic of the reachability graph. Figure 4-3 is RGP output for the Figure 4-2 Petri net text file.

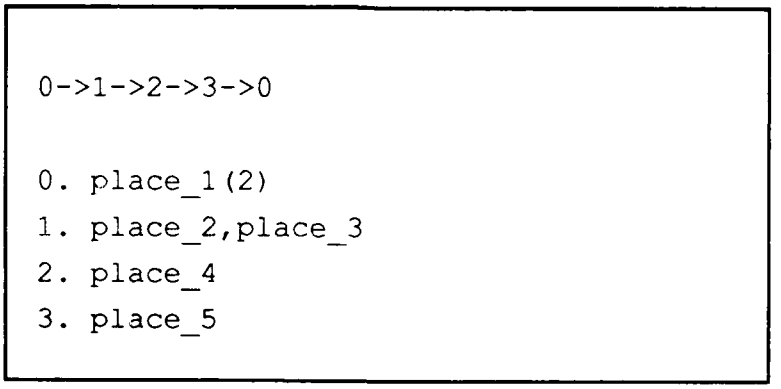


Figure 4-3. **RGP Reachability Graph for Figure 4-1 Petri Net**

In Figure 4-3, notice that the states are numbered from zero to three. The arcs signify which states are reachable from other states and describe all possible state sequences. The marked places comprising each numbered state are listed below the graph.

Although reachability graph printouts and state descriptions proved invaluable in the design and verification of our system component modules, the value of the RGP diminished significantly as complexity and size of the Petri net grew. Our final model had more than 13,000 reachable states and resulted in an RGP output file of over 80,000 lines. Such a large net was impractical to analyze by inspection and required the use of the Reachability Graph Analyzer (RGA).

D. REACHABILITY GRAPH ANALYZER (RGA)

The RGA is a very powerful, interactive interpreter which allows dynamic identifier typing, recursion, and functions. The RGA enables model debugging and proofs of correctness through interactive analysis with the reachability graph. Through the RGA, the user gains access to place names, reachable states, and even the structure of the reachability graph. The RGA functions and capabilities discussed in this section are but a few of those found in Morgan [1987].

To invoke RGA, simply type **rga <file.rg>**. The entered file must be in P-NUT internal reachability graph format.

When the user enters an expression in RGA, the interpreter immediately evaluates it and returns the result. After evaluation, the interpreter discards the previous input and prompts the user for a new command. The prompt symbol is ">". The user can also define expressions and functions for later use.

There are three possible errors that can be encountered while using RGA: syntax errors, run-time errors, and internal RGA errors. Syntax errors result in the message "command ignored" and a prompt. Run-time errors normally result in an appropriate message and a prompt. Internal RGA errors were never encountered in our work.

RGA has a case-sensitive language. Command key words and predefined function names are always written in lower case, while user-defined identifiers may be written in lower, upper, or mixed case. All identifiers must begin with an alphabetic letter and can be followed by zero or more letters, underscores, digits, or periods. A number is one or more digits preceded by an optional minus sign and may be floating point as well as integer.

The RGA interpreter normally recognizes the end of a command by the End Of Line character (EOL). If an expression or function definition is too long for a single line, the use of a reverse diagonal character "\" followed by EOL is treated as a space character. Multiple spaces and tabs are

interpreted as a single space. Comments are signified by enclosure in "/* */", i.e., /* this is a comment */.

Although the expressions and functions in RGA language are evaluated to many different types, our work used only integers, states, Booleans, and sets. If an identifier is assigned a value of any of these types, it will take on the appropriate type.

The value of a place is evaluated as the integer number of tokens it contains in a specified state. To specify the state, the place identifier must be followed by a state-valued expression in parentheses.

State constants are written as a number sign "**#**" followed by an integer. The first state in a reachability graph is denoted in RGA as #0. Places are referenced by the name given in the original input text file used for eventual reachability graph construction. Unnamed transitions are referenced by the dollar sign "$". The RGA standard is to reference the initial transition as the number "0" transition, signified by $0. We found that naming places and numbering transitions (beginning with number zero) enhanced readability and ease of analysis with RGA.

A state in the reachability graph consists of the markings of all places in that state and the sets of arcs to and from predecessor and successor states. The **showstate** function displays the marking of all places in a given state.

Only places that contain one or more tokens are considered to be marked. If a place is marked with more than one token, the number of tokens, enclosed in parentheses, follows the place name.

The most powerful RGA language type we used was sets. Elements of an RGA set must be of the same type and are maintained and displayed by RGA in ascending numerical order. This display feature greatly aids readability and analysis. The predefined set variable used exclusively in our work was S, the set of all reachables net states. Set constants are written as a comma-separated list of states and are enclosed within curly braces, i.e., **{#0,#2..#5}**. This example set consists of the initial state and the sequence of states two through five. If the set is empty, it will be displayed as an empty set of curly braces "{}".

The capability to construct and display subsets of the reachable state space proved invaluable in our research. The method for creating a subset is to specify the parent set followed by Boolean selection criteria. After expression evaluation, RGA will display all elements of the set meeting the Boolean criteria. Uncapitalized s is the predefined subset variable of the set of all reachable states S. The syntax for creating and displaying a desired subset s of S which meets a specific Boolean requirement is **{s in S | <boolean-expression>}**. RGA will evaluate this expression

by calculating and displaying all elements of the subset. To view the place marking of any state, simply use the pre-defined function **showstate** (#<state>).

Variable assignment can be used to store the value of a desired subset. The assignment operator is the colon followed by an equal sign, ":=". Assignment allows the user to later recall the current value of the variable by simply typing the variable identifier following the RGA prompt. RGA will print the current value to the the standard output device.

Available in-fix Boolean operators are standard arithmetic comparison tests: **<**, **<=**, =, **>**, **>=**, and **<>**. While the equal and not equal tests can be applied to any of the data types, the other operators apply only to integers, floating point, and strings. Boolean expression operators **and** and **or** are also included in RGA. The Boolean operator we used most frequently was **exists**, the existential qualifier. An example of the syntax of this operator is: **exists <id> in <set-expression> [<boolean-expression>]**. RGA evaluates the expression by looping through each element of the set expression until it either evaluates an element as true and halts or checks all elements of the set and returns false.

RGA contains several predefined functions. Two functions with substantial safety analysis value are **succ(state)** and

pred(state). **succ(state)** and **pred(state)** respectively return the sets of immediate successor and predecessor states for a specified state. RGA also supports user defined functions. Morgan [1987] contains an example user-defined function which successfully uses **succ** to determine reachability of a given state from any other specified initial state **reachable (initial-state, final-state)**. We altered the code of this function slightly by substituting **pred(state)** for **succ(state)** and produced a working backward reachability function.

The RGA language is highly extensible through its support of user-defined functions and function libraries. Libraries can be created as text files and entered by typing the file names following the reachability graph file name at RGA invocation. An example of an RGA invocation that will include two such libraries is: **rga <file.rg> <function_library1> <function_library2>**. RGA will accept the predefined user functions in these libraries and allow their use in the interactive analysis process. Although we did not use functions in our preliminary investigation, we highly recommend that their capabilities and usefulness be investigated in future research.

V. THE SYSTEM UNDER ANALYSIS

A. A SOFTWARE-CONTROLLED REAL-TIME SYSTEM

The real-time system under analysis is a interrupted-explosive train safety-arming (SA) device for an air-to-air guided missile. The system was the first attempt by the Naval Weapons Center in China Lake, California, to replace a mechanical safe separation distance calculator with a microprocessor and software. The motivations for the conversion are the potential for greater accuracy, tremendous cost reduction, programmability, and the desire to apply state-of-the-art technology to fuzing design.

B. SYSTEM BACKGROUND

Hayward [1987] contains an excellent introduction to the system under analysis. The following discussion is a brief review of that introduction.

A safety arming device is a precision system which incorporates mechanical, electronic, and explosive components. The purpose of the device is to arm the warhead at the correct time in tactical use and to prevent inadvertent high-explosive warhead detonation. The device must operate with high precision and be able to function correctly for the logistic life cycle of the weapon. [McVay, 1987]

MIL-STD-1316C states that a safety arming device must independently prevent unintentional arming and provide forces to remove safety features originating from other environments. At least one of these features must depend on sensing the post-launch environment. The system must also provide an arming delay to ensure safe separation distance is achieved in all defined operational conditions.

In mechanical SA devices, the system is locked in the safe position until unlocked by application of current to a solenoid. The device contains a setback weight, connected by the gears of an escapement mechanism to a rotor. The rotor contains the interrupt element. When the missile's rocket motor fires, the acceleration boost drives the setback weight and causes the movement of the gears. The escapement mechanism serves as a pseudo-integrator to enable movement of the rotor from the interrupted (safe) position to armed after the missile has traveled a preset distance from the launch vehicle. Figure 5-1 shows the block diagram for a standard guided missile SA device [McVay, 1987]. The interruptors have mechanical and direct locking as required by MIL-STD-1316C.

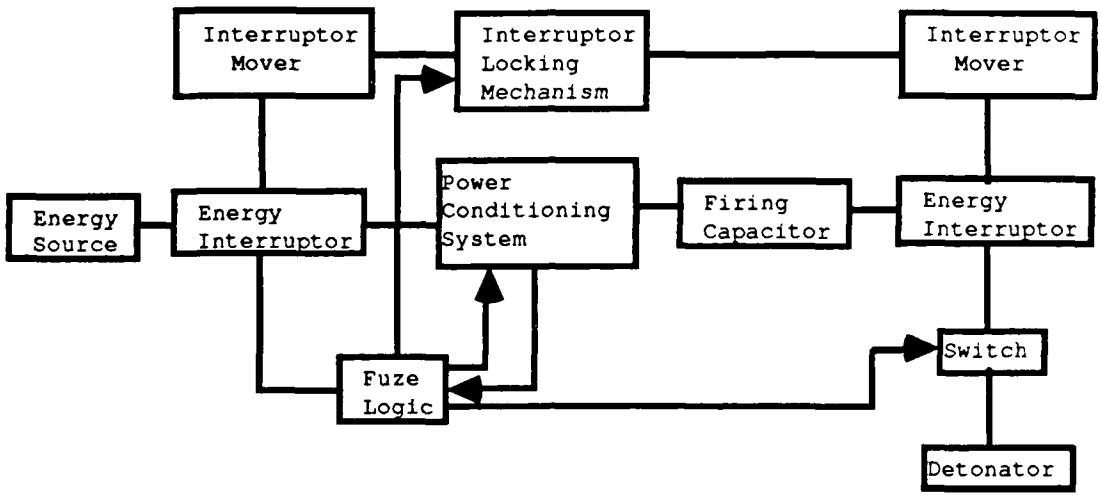


Figure 5-1. **Noninterrupted Explosive Train SA Device**

C. SYSTEM OPERATION

Figure 5-2 is a system flowchart for the SA device under analysis. [Hayward, 1987]

In Figure 5-3, we modified the original Hayward [1987] software flowchart by abstracting the hardware/software control port interface and removing the assembler code identifiers.The firing sequence in Figures 5-2 and 5-3 originates with the missile on an aircraft rack. An intent to launch (ITL) signal initiates a sequence which fires the thermal battery, charges the firing capacitor, powers the computer, and unlocks the SA device. The software then builds a preset safe separation distance lookup table for current distance comparisons.

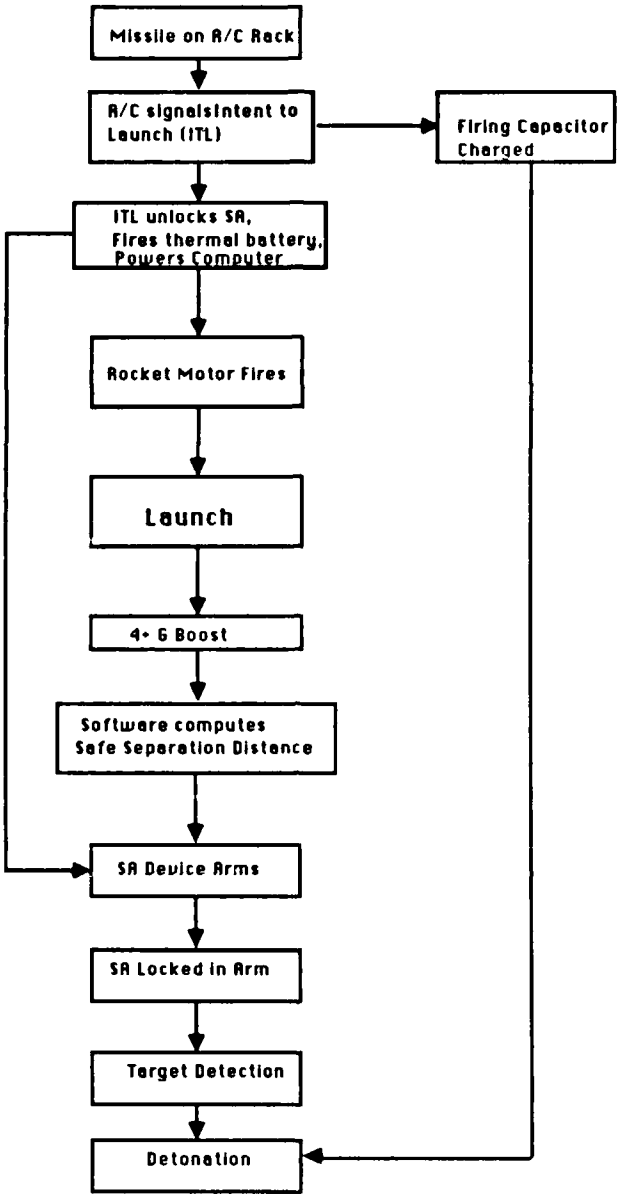


Figure 5-2. **SA Device System Flowchart**

After missile launch, the software uses inputs from an analog to digital converter (ADC) and a timing loop to compute current separation distances from the launch vehicle.

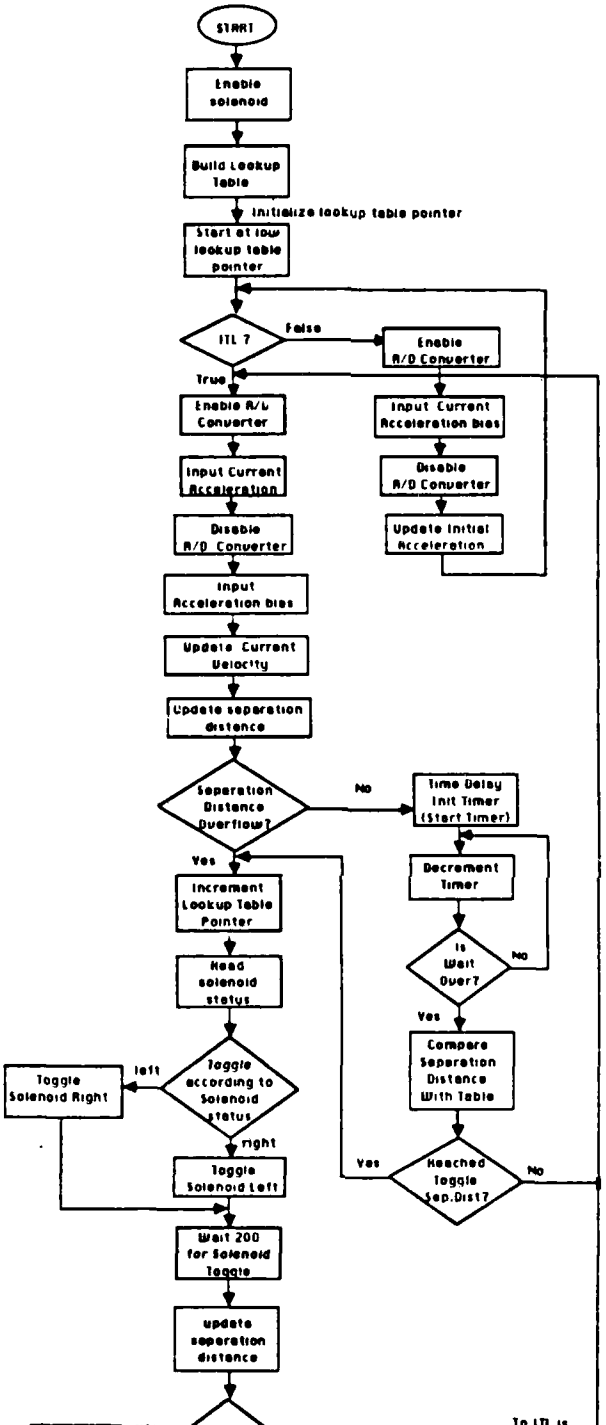


Figure 5-3. **SA Device Software Flowchart**

A minimum 4G boost is required before the program will compare the calculated separation distance with the lookup table values. If the calculated value exceeds the current tabular value, the lookup table entry pointer is advanced to the next table value. The software then commands the solenoid to toggle, resulting in a ball lock mechanism rotating the interruptor by a one-third increment. Next, the software enters a delay loop to provide sufficient time for the solenoid to toggle. The program then loops back, updates the acceleration bias from the ADC output, and starts over. If the calculated value is less than the lookup table entry, the software delays, updates the calculated separation distance, and conducts a second comparison. If the tabular distance is then exceeded, the solenoid is toggled as previously described.Three solenoid toggles are required to remove the interruptor. In addition, the warhead can not detonate unless the SA device is unlocked (which occurs after ITL is signalled) and the firing capacitor is charged.

VI. MODELING AND ANALYSIS METHODOLOGY

A. PROBLEMS IN SOFTWARE SYSTEM MODELING

The major problem in software system modeling is that the model must be sufficiently accurate and detailed to provide meaningful safety analysis results. The model must incorporate the software flowchart, important environmental features, the nature of system components, and any initial conditions. Modeling should be a process of cooperation and continuous feedback between the designer and the modeler. Since the modeling process is difficult, nonessential details should be omitted. Although it is quite difficult to determine detail significance, the reduction of the system scope is important for minimizing required modeling time. If any system facet's significance is unknown, it is best to incorporate it into the model. The system designer must provide feedback to the modeler to ensure a sufficiently accurate model.

Hayward [1987] presented a methodology for building Petri net models of real-time, software-controlled systems. He provides detailed instructions for translation of software flowcharts to Petri nets and discusses a methodology for combining hardware and software system functions into a single Petri net model.

While preparing the Hayward system model for automated analysis, we discovered several modeling flaws and corrected them. Our work reflects those improvements.

B. A BOTTOM-UP APPROACH

Our initial research plan was to familiarize ourselves with the the fuzing system, convert the Hayward [1987] model to entry text file, and conduct P-NUT aided automated safety analysis. Although the Hayward [1987] model was an excellent first attempt, serious shortcomings were soon discovered in the system component models. Following this discovery, we expanded our plan to include corrections to the Hayward model.

After familiarization with the software and components of the safety arming device, we accepted the basic system framework and began at the module level of the Hayward model. We examined the functionality of the existing Petri net modules and compared this with our knowledge of actual component behaviors. This is not the method we recommend for conducting first-time modeling and analysis of a system. As stated in Hayward [1987], the initial modeling process should be a top-down approach, beginning with system and software flowcharts and interfaces, and abstracting out internal component functions. The final step prior to safety analysis should be component modeling and verification. Since we were given an essentially correct system framework, we began our

work from the bottom up. We redefined component interfaces, created the internal component Petri nets, and verified correctness with P-NUT.

1. ITL Sensor Module

We studied the Intent To Launch (ITL) sensor first. The Hayward model is shown in Figure 6-1.

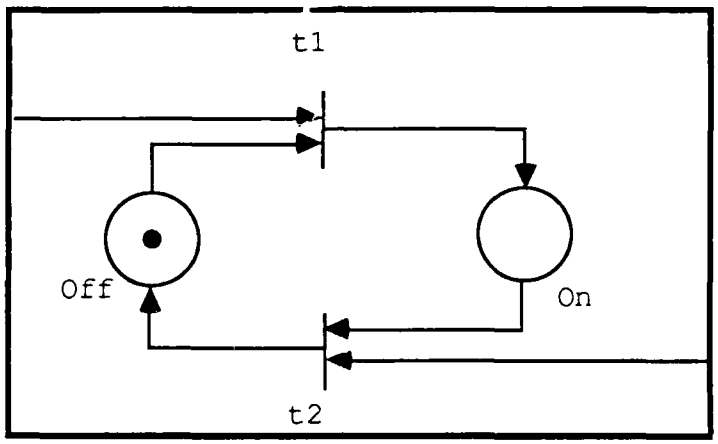


Figure 6-1. **Hayward ITL Sensor Model**

Figure 6-1 effectively models a two-state device, but an ITL sensor must do more than simply toggle. The sensor must have a means for outputting its current state. In the SA device under analysis, the software program checks the ITL sensor to determine which of two control flow paths to follow. The model in Figure 6-1 has no mechanism to allow NonDestructive ReadOut (NDRO) of its stored state and clearly needed this addition.

A component model must reflect system interface requirements and accurately represent behavior of the actual device. As an initial step, the modeler should analyze component functionality and document required system interfaces. He must then ensure that the model accurately represents all significant aspects of function and control flow.

After adding NDRO capability to the ITL sensor, we realized that a proper model of a multifunction device requires a system lock. The lock ensures that once the component receives a command, it prevents new command processing until completion of the original input command. In the ITL sensor, this is critical. Without a system lock, NDRO could occur while the device was toggling and result in erroneous ITL indication. The diagram for the revised ITL sensor is shown in Figure 6-2.

To increase readability and connectivity of the Petri net diagram, we recommend a standard input/output convention for all system modules. This convention is reflected in Figure 6-2 and consists of distinctly shaded input and output places.

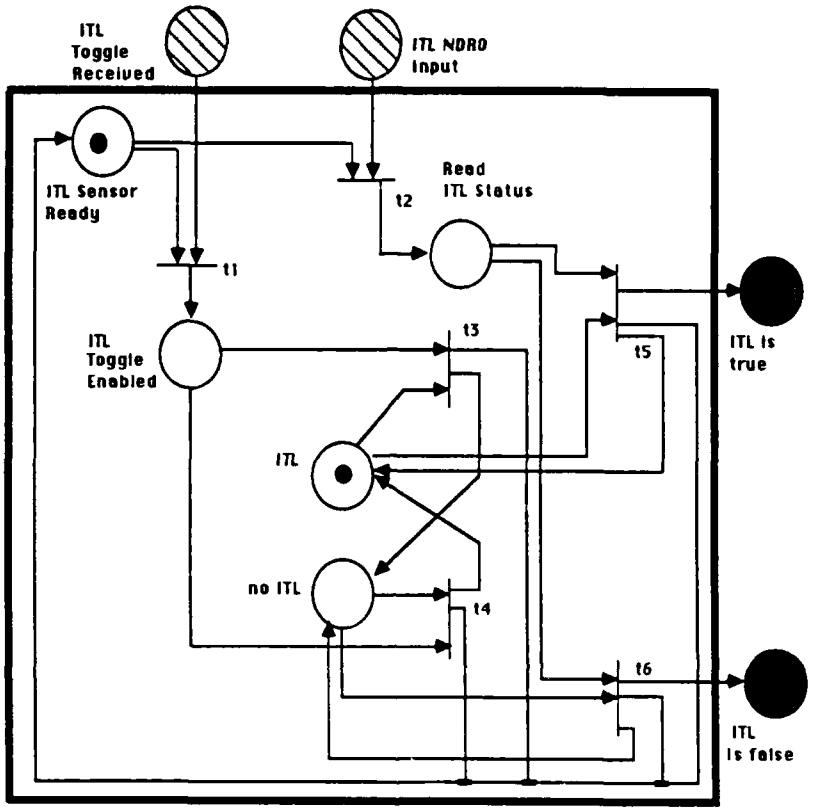


Figure 6-2. **Refined ITL Sensor Model**

In the complete system net, the contents of the modules should be abstracted. The modules are depicted as "black-boxes" with only interface places shown. This convention encourages regular component and system modeling.

Regularity is important due to the nontrivial nature of real-time systems.

In large systems projects, modeling teams could be employed. In these projects, it is both appropriate and necessary to apply software engineering methods for module specification, namely clearly defined and consistent module interfaces.

Beginning with the ITL sensor module in Figure 6-2, we adapted the electrical engineering wiring schematic convention to our modeling of intersecting Petri net arcs. Transitions t3, t4, t5, and t6 create and place tokens in the system-ready place. To conserve space and improve readability, we use intersections with nodes to denote common arcs between several transitions and a single place. It is important to note that this convention is inappropriate for representing arcs between multiple places and a single transition, as this would violate standard Petri net conventions. Additionally, the number of transition inputs and outputs must be readily apparent to an analyst without requiring a count of scattered intersecting lines.

Creation of the ITL sensor in Figure 6-2 was a multi-step process. We began with the Hayward model of a two-state device and through a trial-and-error approach developed the model with NDRO capability.

The fundamental idea of a nondestructive read is to allow the sensor to output its state without changing that state. In Petri net terminology, the device must output state indication and simultaneously return to the marking held prior to the commanded read. Since no modeling of control or information flow is possible without token consumption and creation, the modeler must be innovative but should resist the temptation to build a baroque structure. The addition of places in a net can significantly add to reachability state space size and correspondingly increase analysis difficulty. (There are exceptions to this statement, such as in the use of interlocks, which actually restrict the reachable state space.)

After creating the ITL sensor model, we converted the Petri net to a textual file, built a reachability graph of the system using P-NUT, and analyzed the reachable states by printing the graph. The textual Petri net for the ITL sensor model is shown in Figure 6-3.

To reduce file size, we shortened most place descriptions. We follow this procedure throughout our work. In more complicated Petri nets, ten or more marked places per state are common, often filling several output lines for each state description. Modelers must be uniform selecting place name abbreviations. The **pnl** tool is extremely useful for

uncovering notational discrepancies and should be used prior to translating all text files to internal P-NUT format.

```
:t1: ITL_toggl_rcvd,ITL_snsr_rdy ->ITL_toggl_enabld
:t2: ITL_NDRO_inpt,ITL_snsr_rdy -> rd_ITL_status
:t3: ITL_toggl_enabld,ITL -> no_ITL,ITL_snsr_rdy
:t4: ITL_toggl_enabld,no_ITL -> ITL,ITL_snsr_rdy
:t5: rd_ITL_status,ITL -> ITL_is_true,ITL,ITL_snsr_rdy
:t6: rd_ITL_status,no_ITL -> ITL_is_false,no_ITL,ITL_snsr_rdy

/* following code is for test looping purposes only */

:t7: ITL_is_true -> ITL_toggl_rcvd
:t8: ITL_is_true -> ITL_NDRO_inpt
:t9: ITL_is_false -> ITL_toggl_rcvd
:t10: ITL_is_false -> ITL_NDRO_inpt
<ITL_NDRO_inpt,no_ITL,ITL_snsr_rdy>
```

Figure 6-3. **Textual ITL Sensor Petri Net**

To ensure all reachable states were identified, we added a looping structure at the end of the input text file. These added transitions simply feed the output back into all possible input places, ensuring all inputs and outputs are possible. This procedure is recommended for testing any module that has multiple inputs and outputs. The same effect could be achieved by creating a separate text file and reachability graph for each possible initial condition, however our approach accomplishes this with one text file and reachability graph.We translated the ITL sensor textual net using **transl**, built the reachability graph with the RGB, and used RGP to print it in readable form.This reachability graph is shown in Figure 6-4.The ITL sensor graph and state space

were sufficiently small to allow verification by hand tracing and inspection, thus the RGA was not used.

```
0->1->2->0
       |
       +->3->4->5


0.  ITL_snsr_rdy,ITL_NDRO_inpt,no_ITL
1.  rd_ITL_status,no_ITL
2.  ITL_snsr_rdy,no_ITL,ITL_is_false
3.  ITL_toggl_rcvd,ITL_snsr_rdy,no_ITL
4.  ITL_toggl_enabld,no_ITL
5.  ITL_snsr_rdy,ITL
```

Figure 6-4. **ITL Sensor Reachability Graph/State Space**

2. Analog to Digital Converter (ADC) Model

The Hayward [1987] model for ADC, Solenoid, and Solenoid Control devices is shown in Figure 6-5.

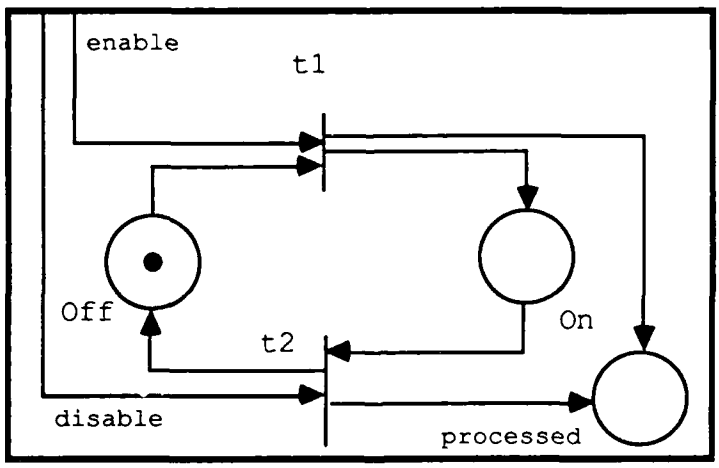


Figure 6-5. **Hayward Two-State Device Model With Response**

Unfortunately the model in Figure 6-5 inadequately represents the actual devices. Our methodology for modeling the ADC follows.

First, we analyzed the function of the ADC in our SA device. The ADC converts an analog input signal (from accelerometers) to a digital acceleration output. It should provide digital acceleration information when enabled and no output when disabled. The ADC has two interfaces with the overall SA system. It outputs digital acceleration values and provides feedback that it has been enabled or disabled.

To represent control flow, we created a model that could be enabled or disabled and provide necessary feedback following command processing. Our approach was limited in that we did not attempt to model the information flow of the module. We assumed that if the ADC was enabled it would provide correct acceleration information, and if disabled it would not. This significant assumption was made to reduce the scope of the model in view of time constraints. It should not adversely affect the credibility of the analysis. If the ADC malfunctioned and provided incorrect acceleration in excess of the actual value, the separation distance would be overestimated and could result in insufficient safe separation distance at detonation. This is an obvious result and there is little need to expend the extra effort and time required to model it. To ascertain correctness and

reliability of hardware components is not the goal in software system analysis. The safety analyst's concern should be for "design safety" or the effect of the component's behavior within the context of the system environment. Figure 6-6 shows the refined ADC model.

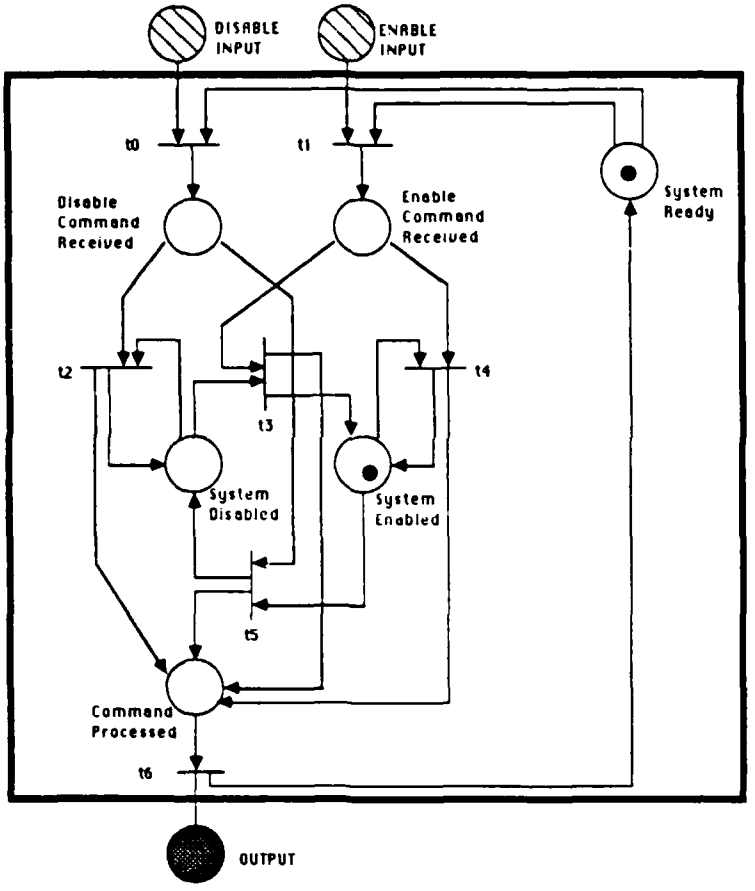


Figure 6-6. **Refined Analog to Digital Converter (ADC) Model**

In Figure 6-6, the system lock, ADC-Ready, reflects device inability to process simultaneous enable and disable commands.

The ADC control structure is more than a simple toggle. It must differentiate between enable and disable commands and allow redundant command processing. This was modeled by adding transitions t2 and t4. If redundant enable or disable commands are received, the model will not change states. It will, however, process the redundant command and signify that it has done so. This is analogous to a component with on and off switches. If the on switch is pressed a second time, it does not cause the system to shut down. If a real-time system has multiple components which can enable or disable a critical component, it normally allows redundant enable/disable commands for insurance purposes. It is our assumption that the ADC is such a device. If redundant commands were not allowed, this could easily be modeled by eliminating the redundant transition's ability to deposit a token in the command-processed place. To accomplish this, simply remove the appropriate arcs. This would eliminate redundant command feedback.

Following model creation, we again turned to P-NUT to verify correctness. We followed the same steps as in analysis of the ITL sensor and produced a printout of the reachability graph and state space. The textual file for the

ADC module is contained in Figure 6-7, while Figure 6-8 depicts the resulting Petri net reachability graph. This was a small listing and analysis began with manual state examination. We examined all reachable states and validated the module. As a final check, we briefly analyzed the module with the RGA.

```
:t0: disable_input, system_ready -> disable_cmd_received
:t1: enable_input, system_ready -> enable_cmd_received
:t2: disable_cmd_received, system_disabled -> system_disabled,
                                              cmd_processed
:t3: enable_cmd_received, system_disabled -> system_enabled,
                                              cmd_processed
:t4: enable_cmd_received, system_enabled -> system_enabled,
                                              cmd_processed
:t5: disable_cmd_received, system_enabled -> system_disabled,
                                              cmd_processed
:t6: cmd_processed -> output, system_ready

/* the following code is for test loop purposes only */

:t7: output -> enable_input
:t8: output -> disable_input

<disable_input, system_enabled, system_ready>
/* initial markings */
```

Figure 6-7. **Textual Version of ADC Petri Net**

Could this model be simultaneously disabled and enabled? We knew this to be impossible from our state inspection and verified it with the RGA. The translation of this question to RGA language is **exists s in S [ADC_enabld(s) = 1 and ADC_disabld(s) = 1].**

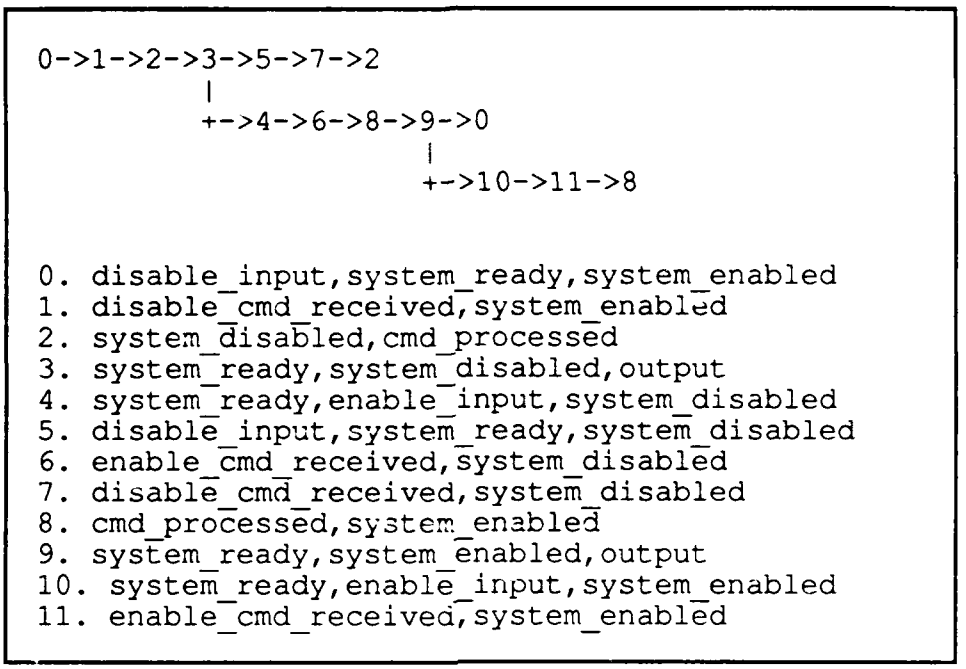


Figure 6-8. **Reachability Graph for ADC Module**

The question is interpreted by RGA as: Is there any reachable state in which there is one token in the ADC enabled place **and** one token in the ADC disabled place? RGA response was false. We then asked the same question using a set variable assignment and assigned the variable name malfunction1 to this particular malfunction. The RGA input for this question is: **malfuntion1 := {s in S | ADC_enabld(s) = 1 and ADC_disabld(s) = 1}.** RGA interpreted this as: Evaluate all elements of the set of reachable states in which both of listed places contain one token and assign this set to the variable malfunction1. RGA responded with a set of empty curly braces, "{}", signifying

that malfunction1 was currently of type set and had the value of null. To redisplay the variable value later in the session, we entered the variable identifier, **malfunction1**, and RGA again responded with the empty set.

We then tested for system deadlocks: **deadlocks := {s in S |nsucc(s) = 0}. nsucc(s)** is a predefined integer expression that returns the number of successor states to a specified state s. The above expression is interpreted by RGA as: Assign to the variable deadlocks all elements of the subset of reachable states that have no possible successor states. The RGA responded with the empty or null set.

3. Solenoid Model

We next turned our attention to the problem of modeling the SA solenoid. The solenoid is a two-state device with enable/disable and NDRO capabilities. Two states refer to status left and status right. When the SA software determines the launched missile has achieved a proper increment of safe separation distance, it checks the solenoid status (right or left) and commands the appropriate toggle. The solenoid toggles, causing the ball lock mechanism to drop a ball and move the fuze interruptor by a one-third increment. Following three toggles, the interruptor is removed. The solenoid system input commands consist of enable/disable, read, and toggle left/right. As in the ITL sensor and ADC, the system accepts only one command at a

time. If a second command is received prior to completion of the first, it is ignored.

Hayward [1987] modeled the solenoid by creating three separate modules: solenoid control, solenoid, and solenoid status toggler. These three independent Petri net models were intended to model a single system component. The solenoid control and solenoid modules were replicas of the two-state device in Figure 6-3. The solenoid control module modeled enable/disable functions and the solenoid module modeled the left/right toggle functions. The third module represented software status indication of the solenoid but toggled independently of the solenoid module.

Since the SA device was never actually constructed, the software prototype required a method to simulate the toggle position. It accomplished this by toggling a status bit in a memory register. The Hayward solenoid status toggler is the model of this register function.

We attempted to analyze safety of the completed software controlled SA device. Accordingly, we modeled the system with all designed software/hardware interfaces incorporated in a single full-function module.

Creation of the solenoid model required several revisions and two weeks of intensive effort. The basic functions were defined and all interfaces specified. We then attempted to abstract another level of the device. Since the

solenoid performed three separate functions, we broke the project functionally into NDRO, left/right toggle, and enable/disable capabilities. Rather than attempt module creation in one step, we combined the ADC "smart" toggle function and the ITL sensor's NDRO capability as our first step. The resulting module is shown in Figure 6-9.

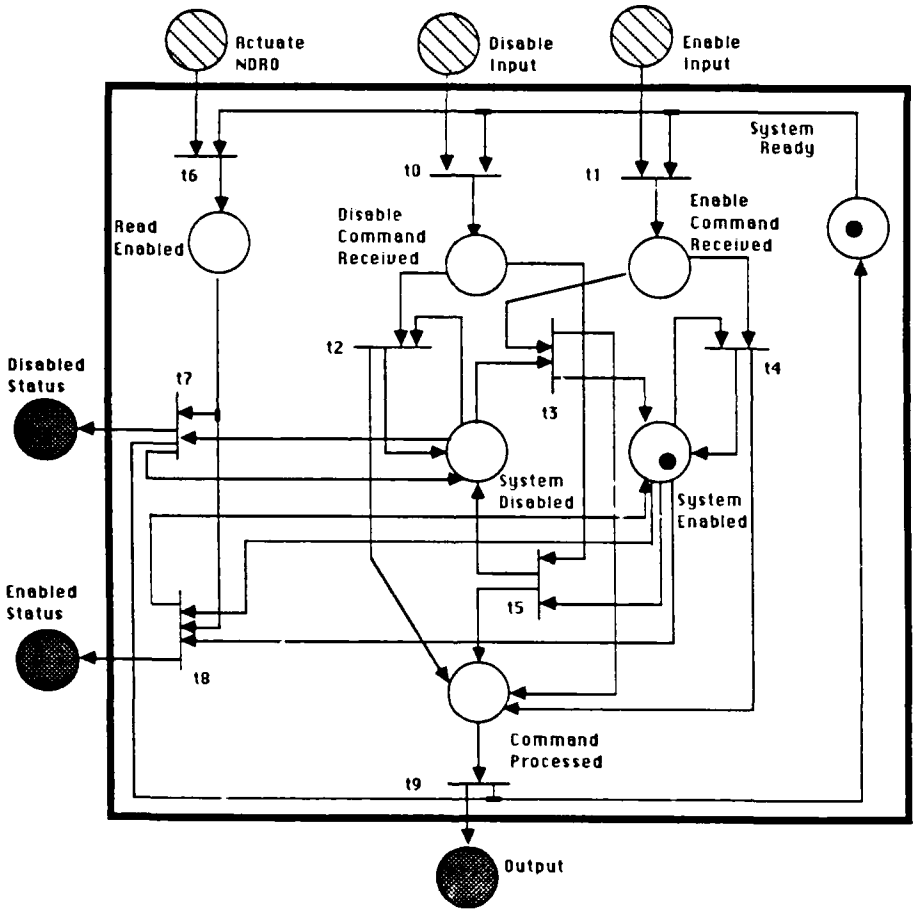


Figure 6-9. **ITL Sensor and ADC Modules Combined**

We then converted the the enable/disable toggle in Figure 6-9 to a left/right function and added another two-state module for enable/disable capability. We added a model feature which would disallow any accumulation of input place tokens. Token accumulation in the model could result from multiple input commands. Since Petri net transitions fire nondeterministically, these input commands might be handled out of order. The actual solenoid has no memory and can not respond to commands received while "in process". In Petri net model terminology, the additional input tokens must be consumed. Figure 6-10 shows the final solenoid model.The solenoid model in Figure 6-10 has three fundamental operating states: ready (enabled), disabled, or in use. System_ready is the equivalent of the system_ready places of the ITL sensor and ADC models in Figures 6-2 and 6-6. A token in this place indicates the solenoid is enabled (has power) and is ready to process input commands. A token in the solenoid_in_use place indicates that the solenoid is processing an input and will not accept further inputs until processing is complete. A marking of the system-disabled place represents power is off and no processing is possible.

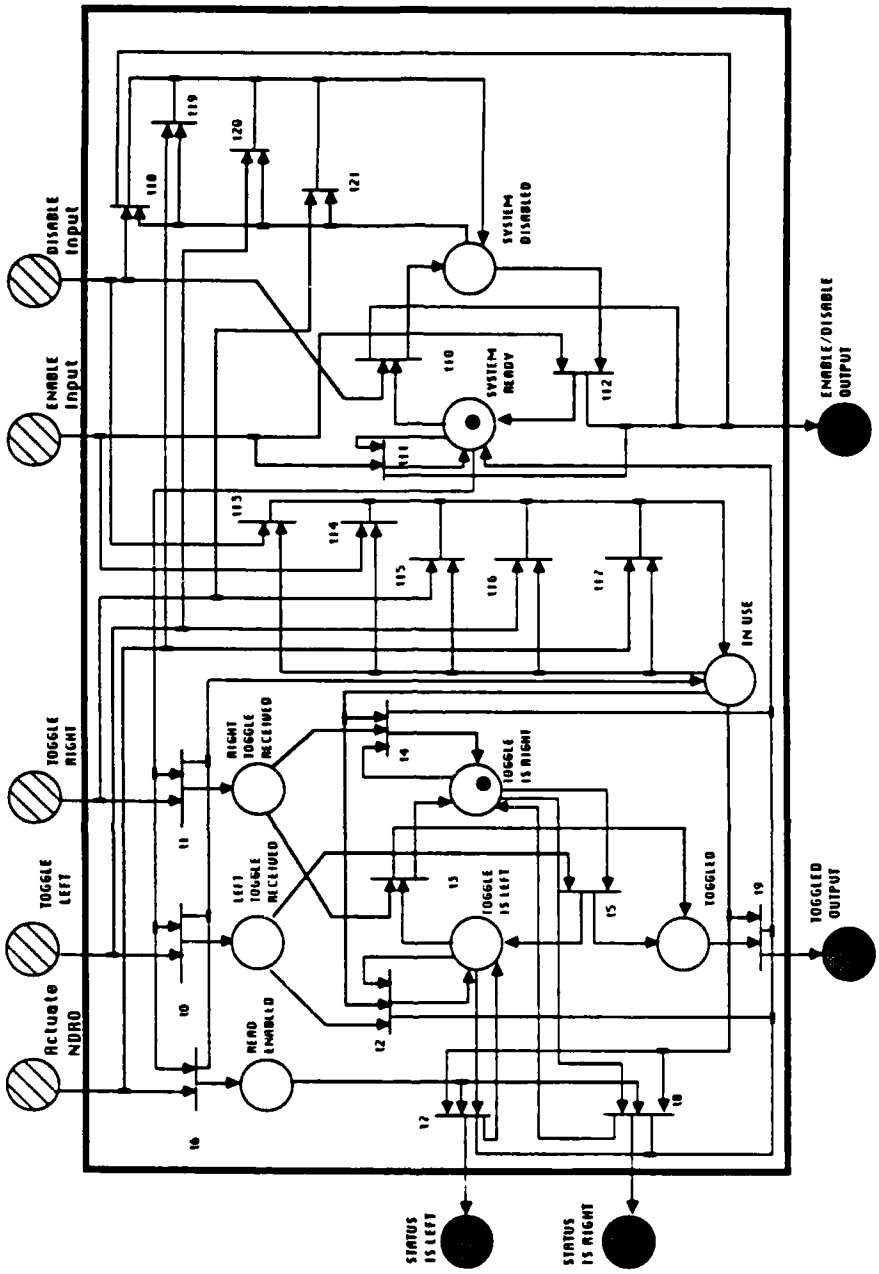


Figure 6-10. **Final Solenoid Model**

The token accumulation problem described above is solved by the addition of token consumption loops. If the solenoid in Figure 6-10 is in use, transitions t13 to t17 ensure consumption of all incoming tokens without altering the command in process. If the solenoid is disabled, transitions t18 to t21 ensure that only an enable command will be processed. All other input tokens will be consumed and the system will remain disabled.

After the solenoid token consumption scheme was incorporated, we did not "backfit" this feature to the ITL sensor and ADC Petri net models. This was solely due to time constraints. Our goal was to demonstrate procedure validity for preventing input token accumulation in any "no-input-memory" device.

In Figure 6-10, no input transitions can fire unless there is an enabling token in the system_ready place. When an input transition fires, this token is consumed and a token is created and placed in the in_use place.

The method for returning the solenoid model to ready state after processing is shown in all the output firing transitions of Figure 6-10. Any transition that fires to a place outside the module requires an input token from thein_use place. When an output transition fires, a token is created and placed in the system_ready place.

A major difference exists between the ADC enable/disable toggle in Figure 6-6 and the solenoid left/right toggle in Figure 6-10. The ADC must treat redundant enable/disable commands no differently than genuine toggles in terms of feedback to the system. (In the solenoid system-ready/disabled toggle, redundant commands are allowed and reflect this philosophy.) Redundant solenoid left/right toggles are not handled in the same manner. If the actual solenoid received a redundant toggle input and provided feedback, no toggle or resulting movement by the interruptor would occur, but system software would erroneously proceed as if these required events had taken place. For this reason, redundant solenoid toggle transition firings do not create a token in the toggled place.

Following model creation, we converted the Petri net to textual form and input it into P-NUT. We translated the file, constructed the reachability graph, printed the graph/state space, and analyzed the graph on RGA. Although larger than the ADC and ITL sensor graphs, the solenoid reachable state listing and graph still permitted manual state inspection and graph tracing. The text file and reachability graph are contained in Appendices D and E.

The final solenoid model in Figure 6-10 reflects a major change from our original module. The error was discovered during solenoid RGA analysis. We asked for the

set of system deadlock states (see Chapter VI) and were surprised when RGA responded with a two-state set. To enhance analysis and prevent deadlocks, we had added transitions to return output tokens back to any input place. One deadlock state had markings in left_toggle_received, toggle_is_left, and in_use, while the other deadlock was the equivalent redundant right toggle state. Redundant left/right toggle transitions, t2 and t4, originally had not required an enabling token from in_use or an output arc back to system_ready. This resulted in the model deadlocking in the in_use condition whenever a redundant toggle was received. This was clearly not our intention. Although we wanted the solenoid model to withhold redundant toggle feedback, we never intended it to result in module deadlock. From a system view, this was equivalent to disabling the solenoid permanently. The correction was easily made and the final text input version is shown in Appendix D. Appendix E contains the resulting reachability graph printout and reachable state descriptions.

We input the corrected model net to RGA. We again asked for the set of deadlocked states and RGA responded with the null set. We then tested for possible accumulation of tokens. More than one token in a given place would signify that the model had failed to prevent command inputs while in use. We changed the input text file (Appendix D) initial

conditions by adding a second token to the toggle_left_input place. A new internal net was created and a reachability graph built. This procedure is required every time changes are made to the input text file. The new reachability graph was then analyzed on RGA. The input expression was **excesstokens := {s in S | tokens(s) >= marked(s)}.** The predefined RGA expression **tokens(s)** is evaluated as the integer number of total tokens present in a given state. **marked(s)** is evaluated as the integer number of places containing at least one token in a given state. If the number of tokens ever exceeds the number of marked places in any state, it indicates token accumulation in at least one place. When RGA evaluates the expression, it returns the subset of reachable states which satisfy the specified Boolean condition. As in previous examples, **showstate(s)** function is then used to display individual place markings within a given state. RGA evaluated our entered expression and returned a set of states in which only the toggle_left_input place contained more than one token. As this was our initial condition, it validated our multiple input token prevention scheme.

We returned to analysis of the model with single token initial markings and asked if it were possible to have the solenoid simultaneously ready and disabled, in use and disabled, or in use and ready. If these supposedly mutually

exclusive markings were possible it would represent a serious design flaw. We labeled this subset identifier as error1. Since we had shown that there was never more than one token in any place, we were able to simplify our input expression. The RGA input for this question was: **error1 := { s in S | (system_ready(s) + system_disabled(s) > 1) or (in_use(s) + system_disabled(s) > 1) or (in_use(s) + system_ready(s) > 1)}**. RGA evaluated the expression and returned the null set. Since we had proven a maximum of one token in each marked place, arithmetically adding the place values in the first half of the Boolean expression was simply a shorthand expression for those states in which any were marked. If any of the places could have contained more than one token, this expression could not be used. The proper expression would then be **error1 := {s in S | (system_ready(s) =1 and system_disabled(s) =1) or (system_ready(s) =1 and in_use(s) =1) or (in_use(s) =1 and system_disabled(s) =1)}**.

We performed a similar test for the erroneous state resulting from any two of the following places simultaneously marked: read_enabled, left_toggle_received, right_toggle_received, system_ready, system_disabled. This expression was entered as **error2 := {s in S | read_enabled(s) + left_toggle_received(s) + right_toggle_received(s) + system_ready(s) +**

system_disabled(s) > 1}. RGA again responded with the null set.

To prove that the module returned to a ready state following output, we asked RGA to return the set of all states in which any of the output status or toggled places was marked simultaneously with the in-use place. We named the subset identifier output_error. The RGA expression of this question was **output_error := {s in S | (status_is_left(s) + status_is_right(s) + toggled_output(s) + en_dis_output = 1) and (in_use(s) = 1)}**. RGA evaluated the expression and returned the null set. This final check completed module correctness validation. To confirm our assessment, we hand-traced the reachability graph and examined all possible state sequences.

4. The System Petri Net Model

Following Solenoid testing, we refined the Hayward [1987] overall system model.

The original Hayward model specified no requirement for missile release prior to the occurrence of a 4G boost. The attainment of this great an acceleration is impossible with the missile on the rack. The interlock between transitions t11 and t12, in Figure 6-11, reflects the necessary sequence of rack release prior to the boost occurring.

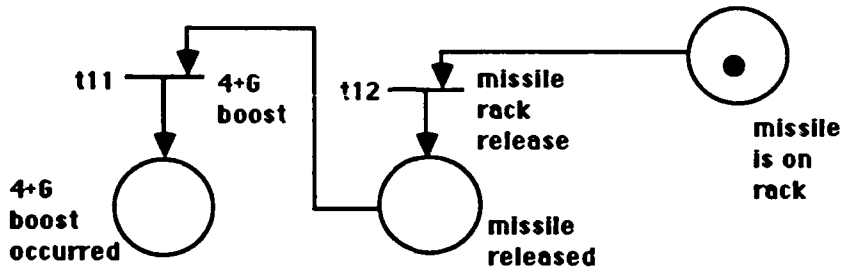


Figure 6-11. **Modeled 4+G Boost Interlock**

In a phone conversation with NWC China Lake, the fuzing system designer, Mr. Steve Rohde, specified a minimum 4G acceleration boost as a precondition to separation distance update calculation. This precondition prevents any toggling of the solenoid or interruptor movement without the required accelerative boost. This feature is modeled in our system Petri net (Appendix F) by making the **4+G boost occurred** a required input place to the **update separation distance** transition, t59. **4+G boost occurred** must be an output place of t59 to ensure accurate reflection that this precondition has been met in any subsequent separation updates.

In Figure 6-12, **ITL locks 1** and **2** and **no-ITL locks 1** and **2** were added after conversion of the net to P-NUT textual input form. While entering the net, we discovered that there was no current method to describe and maintain which of several paths was being taken on ADC entry and exit. In the system control flow, the ADC is enabled following a

read of ITL status. In Figure 6-11, if ITL is true, control flow follows the left path, and if false, the right path is traversed.

If ITL is true, the ADC is enabled, current acceleration is input to the software, and the ADC is disabled. After the ADC is disabled, control and information flow continue along the ITL path toward possible detonation.

If ITL is false, the ADC is enabled, current acceleration bias is sent to the software, and the ADC is disabled. The control and information flow then updates the acceleration and loops back to recheck ITL status.

Although there are four ADC instantiations in the net, the SA system contains only one physical device. The ADC is enabled/ disabled via one set of input places and feedback to the system is via a single set of output places. Since a Petri net is nondeterministic, there must be a modeling methodology to ensure correct path maintenance. The method we devised is to place system locks at module entry and exit points whenever there was a path choice. This lock guaranteed that only the correct transition would be enabled upon module output. Figure 6-12 shows the ADC portion of the final system net with path locks in place.

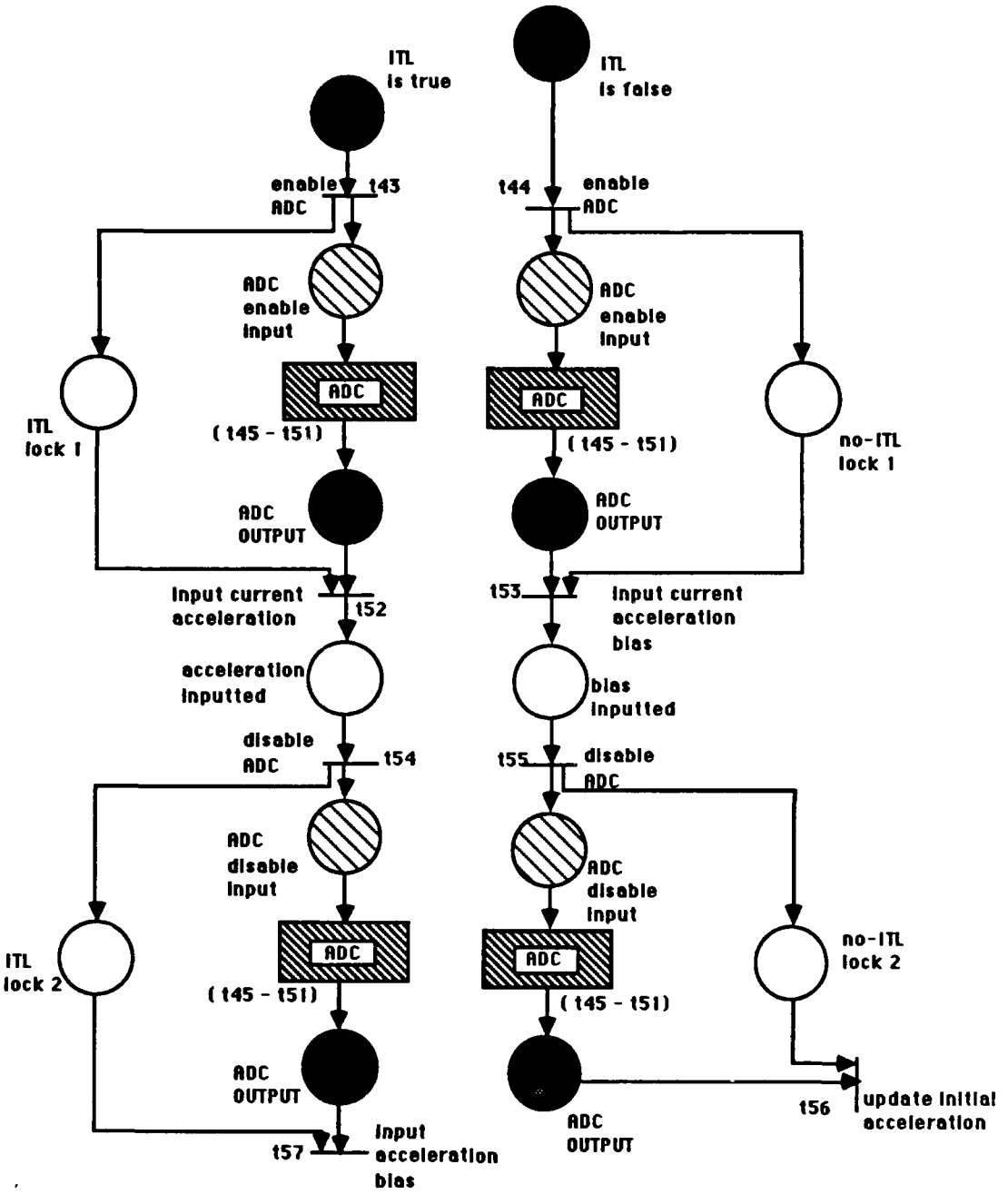


Figure 6-12. **Portion of Final Petri Net Model With Path Locks**

Since three solenoid toggles will result in removing the interruptor, we have used the model proposed in Peterson

[1981] to count the three required software commanded toggles (**iteration counter**). This counter models the system software trap and halts the program following the third software-commanded solenoid toggle. In Figure 6-13, the presence of two tokens in the counter reflects first iteration completion prior to transition t79. Accordingly, this counter will enable t79 for only two firings.

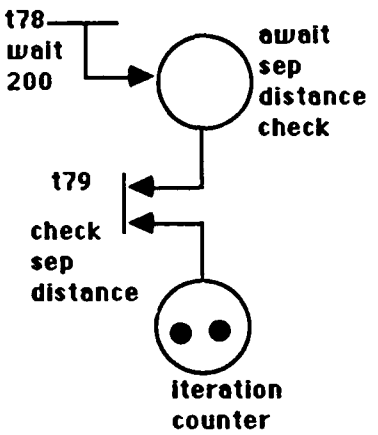


Figure 6-13. **Petri Net Model of a Counter**

There is an aspect of the actual system that we intentionally omitted to reduce model scope and complexity. Actual system software enables the solenoid prior to, and disables it following, each read/toggle. We enable the solenoid initially and do not disable it on each iteration. This should not affect model validity, since we proved earlier that the disabled solenoid model could neither toggle nor output status.

5. P-NUT Aided Safety Analysis of System Model

We converted the graphical net to P-NUT text file format, translated it, built the reachability graph, and printed the reachability graph to an output file. The resulting RGP output contained over 13,000 states and required over five megabytes of memory storage. The sheer magnitude of the reachability graph precluded manual examination and forced total reliance on automated RGA analysis. The textual version of the system Petri net is shown in Appendix G.

Due to time constraints, we limited our RGA safety analysis to reachability of major hazardous and mishap states.

To determine the possibility of missile detonation while attached to aircraft wing we entered: **hazard1 := { s in S | missl_on_rack(s) = 1 and detonation(s) = 1}.** RGA responded with the null set, meaning it was not a reachable state of our model.

To determine if detonation could occur when ITL was false, we entered the expression: **hazard2 := { s in S | no_ITL(s) = 0 and detonation(s) = 1].** RGA responded with the null set.

We next determined the possibility of detonation occurring without a minimum 4G boost. We entered **hazard3 :=**

{ s in S | fourG_bst_occrrd(s) = 0 and detonation(s) = 1}, and RGA returned the null set.

As a final question, we asked if detonation could occur if no power was routed to the computer. The expression **hazard4 := { s in S | cmptr_off(s) = 1 and detonation(s) = 1}** also resulted in the null set.

VII. RESULTS AND CONCLUSIONS

A. RESULTS

This thesis has proposed a methodology for Petri net modeling and automated safety analysis of a real-time concurrent system. The sample system is a proposed air-to-air guided missile Safety and Arming (SA) device. The methodologies for initial modeling and safety analysis of this representative real-time system methodology were originally presented in Hayward [1987]. Our goal was to refine the modeling methodology presented by Hayward and to demonstrate the methodology for, and the feasibility of, automating the safety analysis.

We introduced software safety, Petri nets, and reachability theory. We demonstrated steps required to use the Petri Net Utility Tools (P-NUT) and discussed the extensibility of the P-NUT Reachability Graph Analyzer (RGA). We next discussed P-NUT potential for automating detailed safety analysis.

We have presented a methodology for system safety analysis using Petri net modeling. Using this methodology, we initially analyzed all aspects of system functionality and documented internal interfaces. We then abstracted the individual components and constructed Petri net component models based on a thorough study of their operation, control

flow, and system interfaces. We converted each Petri net model to textual form, entered it into P-NUT, and validated model design using P-NUT automated tools. After all system component models were verified, we examined the system Petri net, as presented in Hayward [1987]. After comparing the control flow of the net with our understanding of actual system operation, we discussed the basis for several refinements and demonstrated use of P-NUT to construct the reachability graph. Function and use of various RGA expressions and predefined functions for examination of hazardous state reachability were then presented. We concluded by demonstrating, from a preliminary standpoint, how to determine system safety.

B. CONCLUSIONS

As previously stated, our initial research goal was the automated analysis of a preexisting real-time system Petri net model. Unfortunately, the preexisting system model [Hayward 1987] was incomplete, requiring remodeling of components and refinement of the system Petri net structure. We essentially accepted the Hayward system framework and employed a bottom-up approach. We redesigned component interfaces and internals and verified correctness on the module scale prior to refining the system net structure and conducting automated safety analysis.

The recommended chronology for Petri net modeling and automated safety analysis follows. A brief summary is contained in Appendix H.

Initially, the modeler/analyst should study system function and the designer's description of possible mishap or hazardous states. He must then reduce the scope of the system to include only significant aspects and details pertinent to stated hazards. Although this is possibly the most difficult step in the entire process, it is critical to reducing scope and complexity of the resulting net. By the very nature of mishaps, it is extremely difficult to ascertain the relative import of individual components or processes. It is therefore imperative to incorporate any features of the system that may contribute to reaching hazardous or mishap states. If more model detail is desired after initial net construction, it is possible to refine the original Petri net. Continuous feedback from the system designer is critical to accurate modeling and safety analysis.

The modeler must study system and software flowcharts thoroughly. He must document all software/hardware interfaces and incorporate the flowcharts into a single Petri net system description. Component internals should initially be abstracted with "black-box" descriptions and incorporate only required system interfaces. Multiple instantiations of

a component should be presented in the system net as separate "black-box" descriptions.

After the framework is complete and approved by the designer, system component functionality should be studied and modeled. As in the system framework approach, one must incorporate only the desired aspects of component behavior and attempt to further divide the individual components into submodules, i.e., NDRO module, toggle modules, etc. An example of this second level of abstraction is found in our solenoid module (Appendix C). This Petri net model is basically a combination of three two-state devices with appropriate internal interfacing.

As each component model is completed, P-NUT should be used to verify desired behavior. Proving component function and interface correctness on the module level permits easy and accurate incorporation into the system net.

After completing the initial system model, P-NUT should be employed to construct the reachability graph. RGA automated safety analysis is then possible and can verify desired system behavior and safety.

Manual construction of a 13,000-state reachability graph would be an arduous, lengthy, and error-prone process. P-NUT can construct this size graph on a Sun 2 computer in under two minutes of CPU time.

P-NUT is an extremely powerful suite of tools for Petri net analysis. One of the major drawbacks of the Petri net safety analysis methods cited in Leveson and Stolzy [1987] is the difficulty of constructing reachability graphs for complex concurrent-system Petri nets. The Reachability Graph Builder eliminates this problem, while the Reachability Graph Analyzer permits detailed analysis of the graph and reachable state space. The extensibility of the RGA language makes it extremely powerful and allows safety analysts to create and use predefined function libraries. We have shown how RGA quickly evaluates sensible and important safety questions. Answering these questions for complex reachability graphs would be extremely difficult, if not impossible, based on manual construction and analysis.

While conducting our research, we came to fully appreciate the feasibility and suitability of Petri nets for software system modeling and safety analysis. In the course of modeling system components, Petri nets were versatile enough to enable accurate modeling of any system aspect desired. Petri nets captured every essential feature of system interface and control/information flow. Petri net modeling requirements for full specification of system interactions and dependencies forced us to explicitly state all control flow or behavior assumptions. Often, by merely

converting our ideas to a Petri net structure, we observed previously unnoticed assumption irregularities.

Methodologies presented in Hayward [1987] and this thesis show that real-time system safety analysis is feasible using Petri nets and the automated P-NUT suite. We have endeavored to introduce the reader to these techniques and strongly encourage further research in the area.

C. RECOMMENDATIONS

We have attempted to demonstrate the feasibility of applying automated analysis tools to Petri net software system modeling and safety analysis. The methodologies presented are only a preliminary step in creating a complete methodology for successful and accurate real-time system Petri net modeling and automated safety analysis.

We strongly recommend that the next refinement of these techniques incorporate timed Petri nets. The synchronization facets of real-time concurrent systems can only be modeled and analyzed completely if timing constraints are included. Leveson and Stolzy [1987] discuss the application of timed Petri nets to the modeling process. P-NUT contains tools capable of construction and analysis of timed Petri net reachability graphs. [Razouk, 1987; Morgan, 1987]

Leveson [1986] presents an algorithm for determination and elimination of Petri net critical states. A logical next step in automating software safety analysis is conversion of

this algorithm to RGA language. Translation should be possible, given RGA language extensibility.

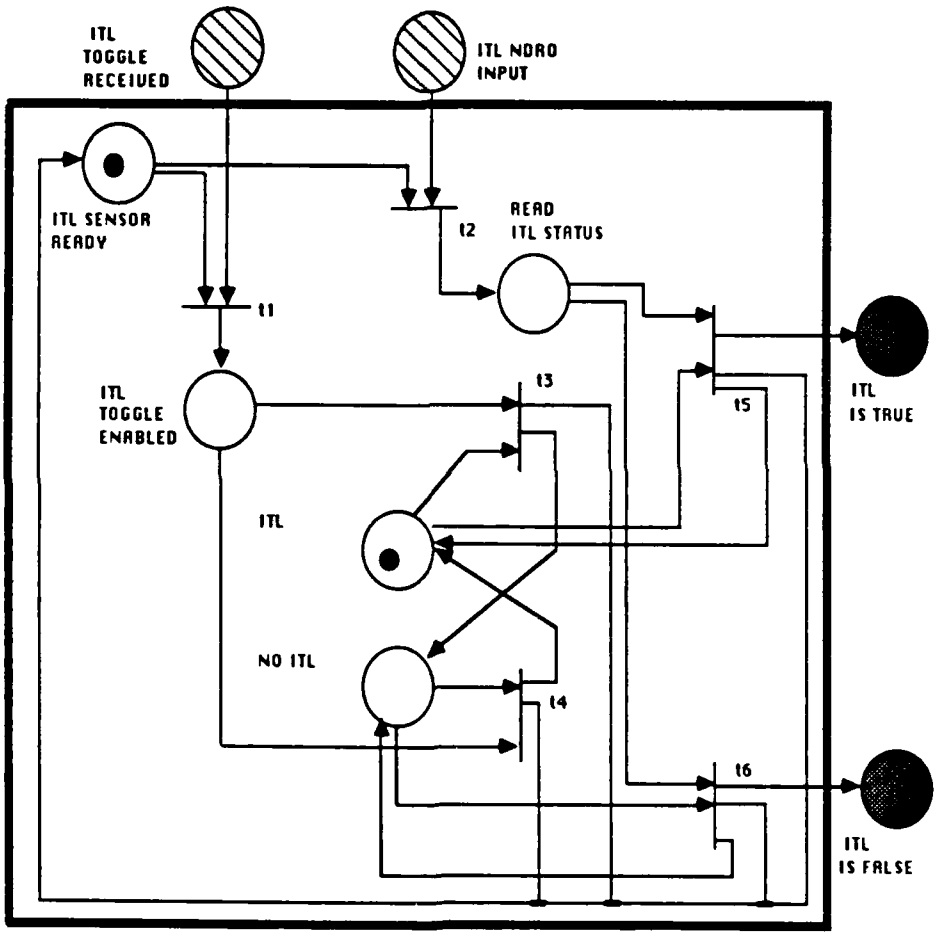
Leveson and Stolzy [1987] introduced the methodology of simulating system faults within a Petri net model. The technique consists of adding fault transitions to cause unintended events or prevent occurrence of intended events. Automated safety analysis is particularly suitable for this technique as a new net reachability graph must be constructed for any Petri net change. Although this is a very difficult manual task, automation enables complex graph creation in several minutes. The analyst could quickly model and analyze a variety of specific component or software malfunctions. This would allow for more complete and accurate assessment of overall system safety.

Gaining familiarity with P-NUT is a difficult process. We strongly recommend the creation of an automated user interface. This interface might allow user construction and storage of a graphical Petri net via P-NUT. The software could use a graph-drawing capability with predefined place, transition, and arc components. It could then cue the user for suitable identifiers and interface with the Reachability Graph Builder and Reachability graph analyzer for translation of user reachability questions. With current user interface technology this capability seems reasonable. P-NUT is a very powerful and effective collection of Petri net analysis

tools. The only drawback to large-scale employment in further safety analysis research is the current awkwardness and difficulty of the user interface.

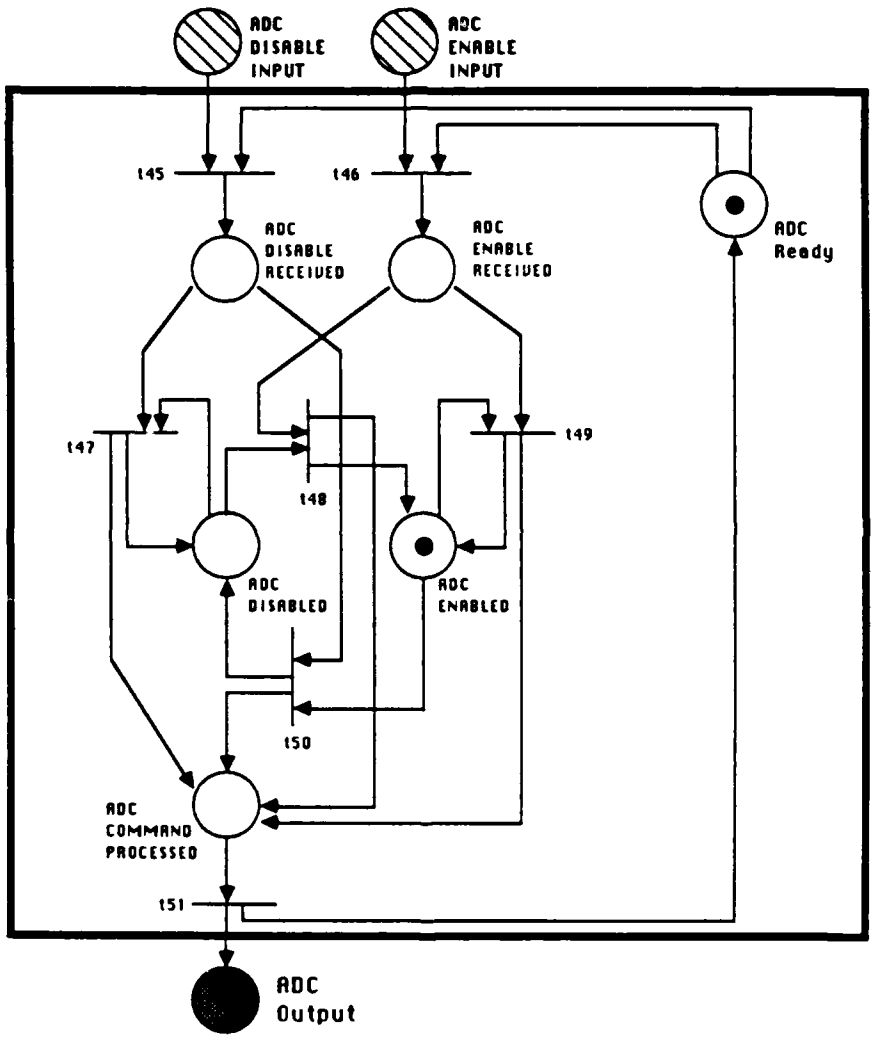
APPENDIX A

INTENT TO LAUNCH (ITL) SENSOR PETRI NET MODEL



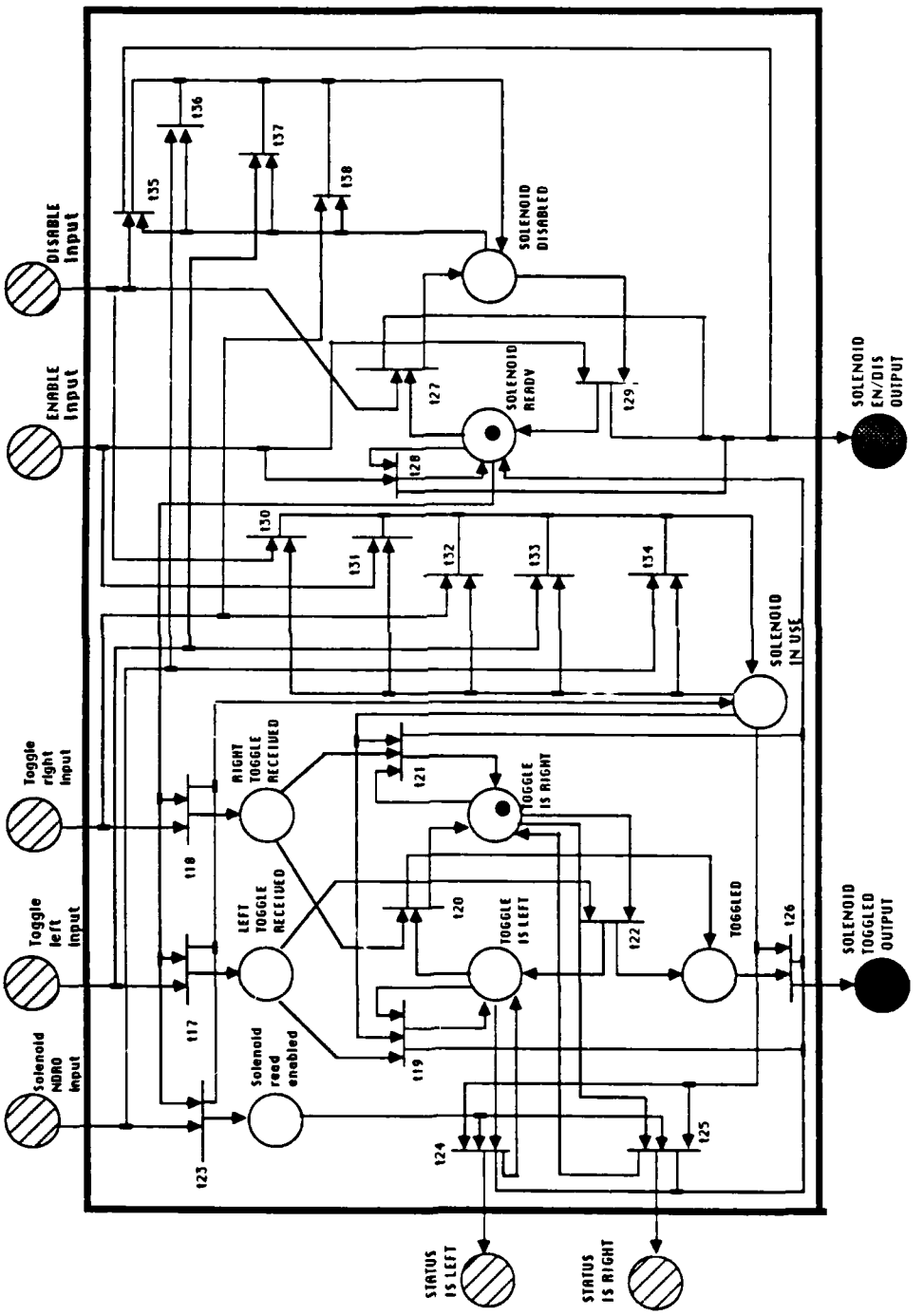
APPENDIX B

ANALOG TO DIGITAL CONVERTER (ADC) PETRI NET MODEL



APPENDIX C

SOLENOID PETRI NET MODEL



APPENDIX D

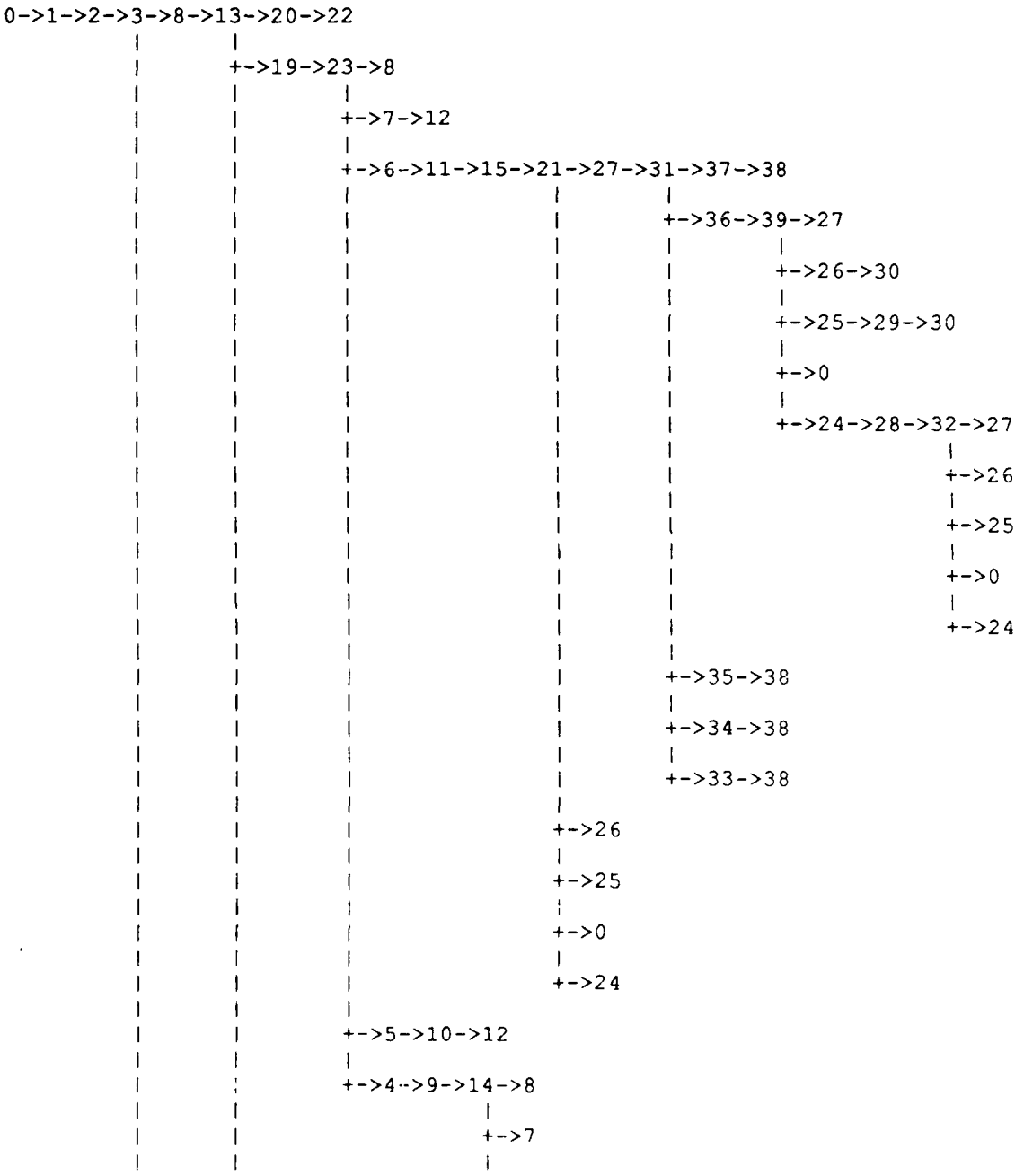
SOLENOID PETRI NET TEXT FILE

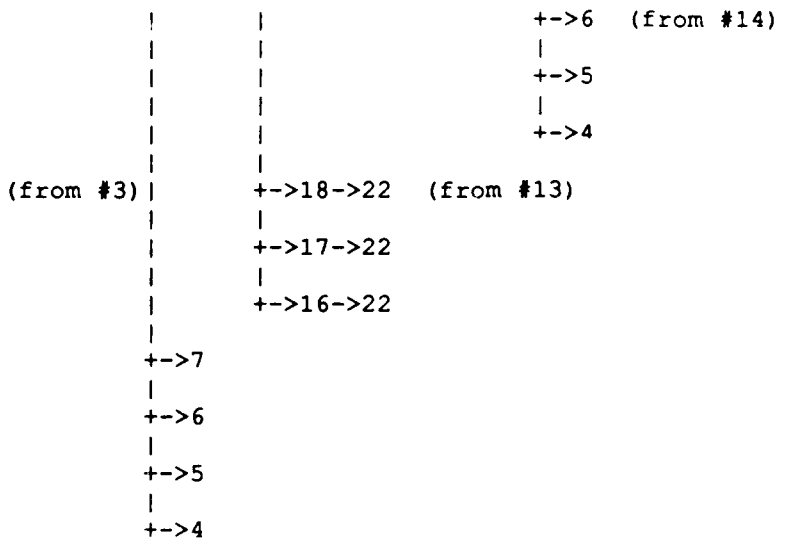
```
:t0: toggle_left, system_ready -> left_toggle_received, in_use
:t1: toggle_right,system_ready -> right_toggle_received,in_use
:t2: in_use,left_toggle_received,toggle_is_left ->toggle_is_left,\
                                                    system_ready
:t3: right_toggle_received,toggle_is_left -> toggle_is_right,\
                                                    toggled
:t4:in_use,right_toggle_received,toggle_is_right, toggled ->\
                                    toggle_is_right, system_ready
:t5: left_toggle_received,toggle_is_right -> toggle_is_left,\
                                                    toggled
:t6: read_status,system_ready -> read_enabled,in_use
:t7: in_use,read_enabled,toggle_is_left -> status_is_left,\
                                    toggle_is_left, system_ready
:t8: in_use,read_enabled,toggle_is_right -> status_is_right, \
                                    toggle_is_right, system_ready
:t9:  in_use,toggled -> toggled_output,system_ready
:t10: disable_input,system_ready -> system_disabled,en_dis_output
:t11: enable_input,system_ready -> system_ready
:t12: enable_input,system_disabled -> system_ready,en_dis_output
:t13: enable_input,in_use -> in_use
:t14: disable_input,in_use -> in_use
:t15: toggle_right,in_use -> in_use
:t16: toggle_left,in_use -> in_use
:t17: read_status,in_use -> in_use
:t18: disable_input,system_disabled -> system_disabled
:t19: read_status,system_disabled -> system_disabled
:t20: toggle_left,system_disabled -> system_disabled
:t21: toggle_right,system_disabled -> system_disabled

/* the following code is for test loop purposes only*/

:t22: status_is_left -> read_status
:t23: status_is_left -> toggle_left
:t24: status_is_left -> toggle_right
:t25: status_is_left -> enable_input
:t26: status_is_left -> disable_input

:t27: status_is_right -> read_status
:t28: status_is_right -> toggle_left
:t29: status_is_right -> toggle_right
:t30: status_is_right -> enable_input
:t31: status_is_right -> disable_input

:t32: toggled_output -> read_status
:t33: toggled_output -> toggle_left
:t34: toggled_output -> toggle_right
```

```
:t35: toggled_output -> enable_input
:t36: toggled_output -> disable_input

:t37: en_dis_output -> read_status
:t38: en_dis_output -> toggle_left
:t39: en_dis_output -> toggle_right
:t40: en_dis_output -> enable_input
:t41: en_dis_output -> disable_input

<toggle_left(1),toggle_is_right(1),system_ready(1)> /*initial
markings*/
```

APPENDIX E

SOLENOID REACHABILITY GRAPH



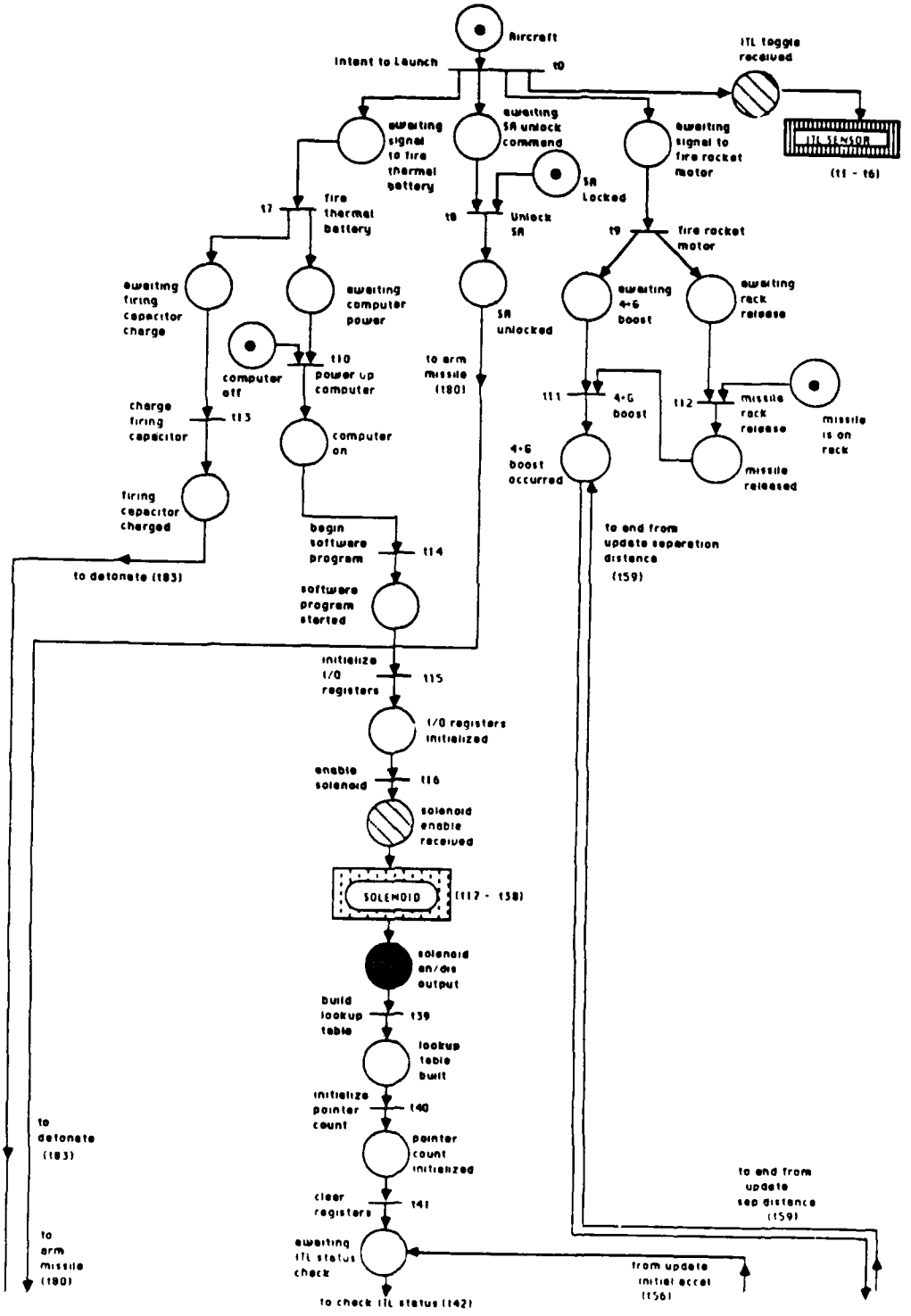


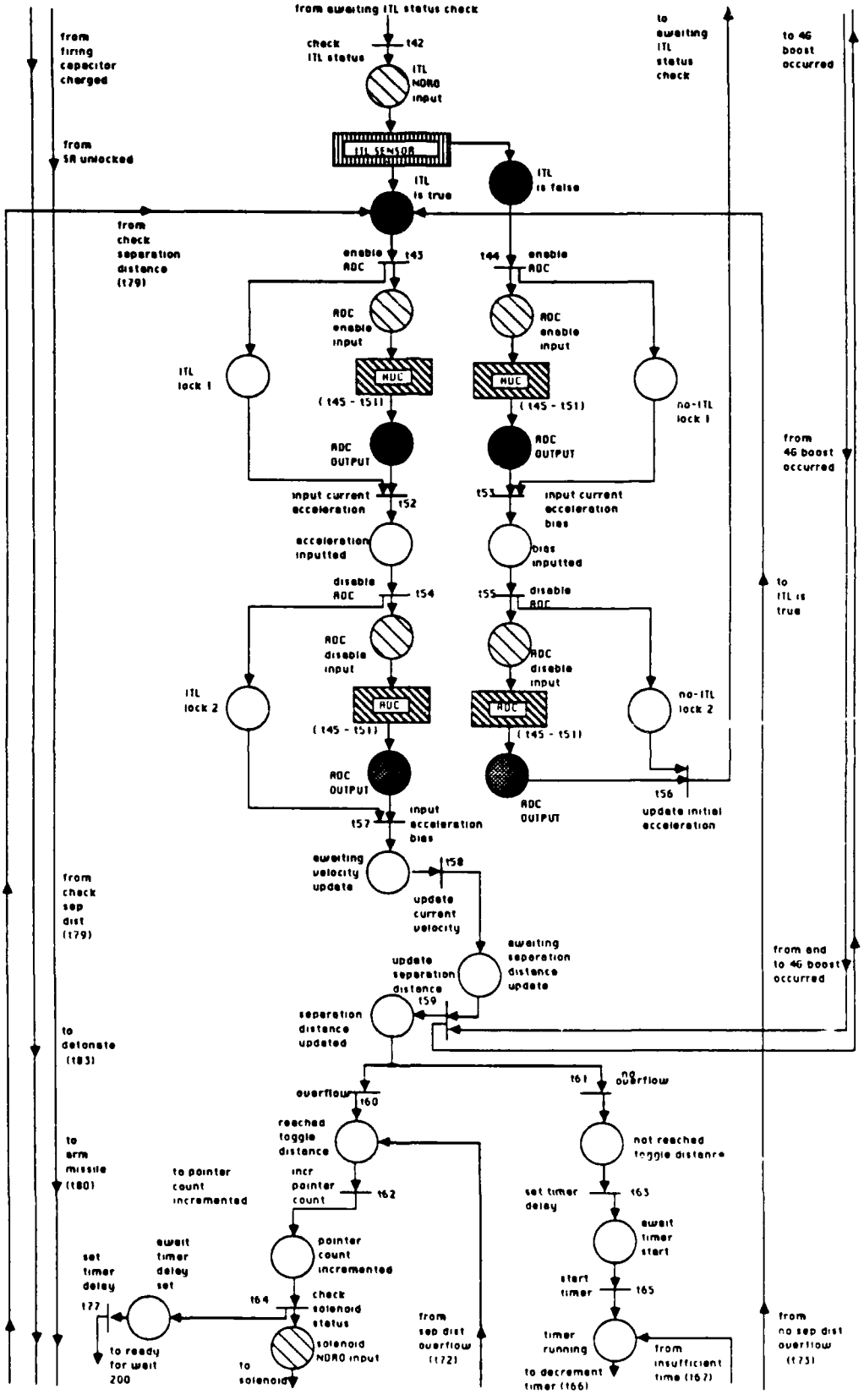
0. toggle_left,system_ready,toggle_is_right
1. left_toggle_received,in_use,toggle_is_right
2. in_use,toggle_is_left,toggled
3. system_ready,toggle_is_left,toggled_output
4. system_ready,toggle_is_left,read_status
5. toggle_left,system_ready,toggle_is_left
6. system_ready,toggle_right,toggle_is_left
7. system_ready,toggle_is_left,enable_input
8. system_ready,toggle_is_left,disable_input
9. in_use,toggle_is_left,read_enabled
10. left_toggle_received,in_use,toggle_is_left
11. in_use,right_toggle_received,toggle_is_left
12. system_ready,toggle_is_left
13. toggle_is_left,system_disabled,en_dis_output
14. system_ready,toggle_is_left,status_is_left
15. in_use,toggle_is_right,toggled
16. toggle_is_left,read_status,system_disabled
17. toggle_left,toggle_is_left,system_disabled
18. toggle_right,toggle_is_left,system_disabled
19. toggle_is_left,system_disabled,enable_input
20. toggle_is_left,disable_input,system_disabled
21. system_ready,toggle_is_right,toggled_output
22. toggle_is_left,system_disabled
23. system_ready,toggle_is_left,en_dis_output
24. system_ready,toggle_is_right,read_status
25. system_ready,toggle_right,toggle_is_right
26. system_ready,toggle_is_right,enable_input
27. system_ready,toggle_is_right,disable_input
28. in_use,toggle_is_right,read_enabled
29. in_use,right_toggle_received,toggle_is_right
30. system_ready,toggle_is_right

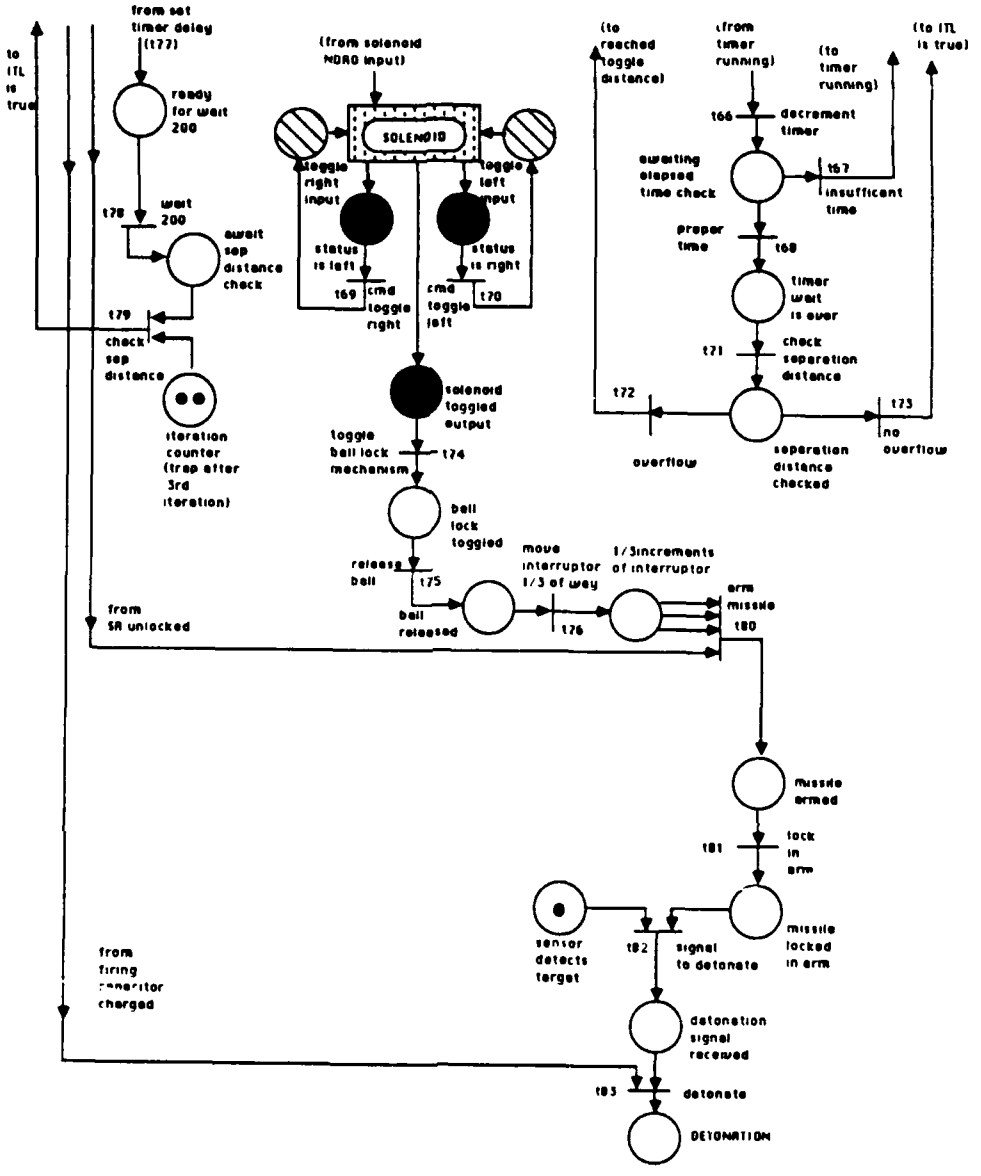
31. toggle_is_right,system_disabled,en_dis_output
32. system_ready,toggle_is_right,status_is_right
33. toggle_is_right,read_status,system_disabled
34. toggle_left,toggle_is_right,system_disabled
35. toggle_right,toggle_is_right,system_disabled
36. toggle_is_right,system_disabled,enable_input
37. toggle_is_right,disable_input,system_disabled
38. toggle_is_right,system_disabled
39. system_ready,toggle_is_right,en_dis_output

APPENDIX F

SAFETY AND ARMING (SA) SYSTEM




from awaiting ITL status check
from firing capacitor charged
from SA unlocked
check ITL status
t42
ITL NORO input
ITL SENSOR
ITL is true
ITL is false
to awaiting ITL status check
to 46 boost occurred
from check separation distance (t79)
enable ADC
t43
t44
ADC enable input
ADC
(t45 - t51)
ITL lock 1
no-ITL lock 1
ADC OUTPUT
from 46 boost occurred
input current acceleration
t52
t53
input current acceleration bias
acceleration inputted
bias inputted
disable ADC
t54
t55
ITL lock 2
no-ITL lock 2
to ITL is true
t56
update initial acceleration
input acceleration bias
t57
from check sep dist (t79)
awaiting velocity update
t58
update current velocity
awaiting separation distance update
update separation distance
t59
from end to 46 boost occurred
separation distance updated
to detonate (t83)
overflow
t60
no overflow
t61
reached toggle distance
not reached toggle distance
to arm missile (t80)
to pointer count incremented
incr pointer count
t62
set timer delay
t63
await timer start
await timer delay set
pointer count incremented
start timer
t65
t64
check solenoid status
t72
from sep dist overflow (t72)
timer running
from insufficient time (t67)
from no sep dist overflow (t73)
to ready for wait 200
to solenoid
solenoid NORO input
to decrement timer (t66)


from set timer delay (t77)
to ITL is true
ready for wait 200
t78
wait 200
await sep distance check
t79
check sep distance
iteration counter (trap after 3rd iteration)
(from solenoid NORO input)
SOLENOID
toggle right input
toggle left input
status is left
status is right
t69
cmd toggle right
cmd toggle left
t70
solenoid toggled output
toggle ball lock mechanism
t74
ball lock toggled
release ball
t75
ball released
move interruptor 1/3 of way
t76
1/3increments of interruptor
arm missile
t80
from SR unlocked
missile armed
t81
lock in arm
sensor detects target
t82
signal to detonate
missile locked in arm
from firing capacitor charged
detonation signal received
t83
detonate
DETONATION
(to reached toggle distance)
(from timer running)
(to timer running)
(to ITL is true)
t66
decrement timer
awaiting elapsed time check
t67
insufficent time
proper time
t68
timer wait is over
t71
check separation distance
t72
t73
overflow
separation distance checked
no overflow

APPENDIX G

SA SYSTEM PETRI NET TEXT FILE

```
:t0: aircrft -> rdy_to_fire_therm_btry,awaitng_SA_unlck_cmd,\
                    awaitng_sig_to_fire_rckt_mtr,ITL_toggl_rcvd

/* following is ITL sensor module, t1 - t6  */

:t1:  ITL_toggl_rcvd,ITL_snsr_rdy -> ITL_toggl_enabld
:t2:  ITL_NDRO_inpt,ITL_snsr_rdy -> rd_ITL_status
:t3:  ITL_toggl_enabld,ITL -> no_ITL,ITL_snsr_rdy
:t4:  ITL_toggl_enabld,no_ITL -> ITL,ITL_snsr_rdy
:t5:  rd_ITL_status,ITL -> ITL_is_true,ITL_snsr_rdy,ITL
:t6:  rd_ITL_status,no_ITL -> ITL_is_false,ITL_snsr_rdy,no_ITL,\
                                awaitng_cmptr_pwr
:t8:  awaitng_SA_unlck_cmd,SA_lckd -> SA_unlckd
:t9:  awaitng_sig_to_fire_rckt_mtr -> awaitng_4G_bst, \
                                      awaitng_rack_rel
:t10: awaitng_cmptr_pwr,cmptr_off -> cmptr_on
:t11: awaitng_rack_rel,missl_on_rack -> missl_rlsd
:t12: awaitng_4G_bst,missl_rlsd -> fourG_bst_occrrd
:t13: awaitng_frng_cap_chg -> frng_cap_chgd
:t14: cmptr_on -> sftwre_prgrm_startd
:t15: sftwre_prgrm_startd -> IO_reg_initlzd
:t16: IO_reg_initlzd -> solnoid_enabl_inpt

/* following is Solenoid module, t17 _ t38 */

:t17: toggl_lft_inpt,solnoid_rdy -> lft_toggl_rcvd,solnoid_in_use
:t18: toggl_rt_inpt,solnoid_rdy -> rt_toggl_rcvd,solnoid_in_use
:t19: lft_toggl_rcvd,toggl_is_lft,solnoid_in_use -> toggl_is_lft, \
                                                    solnoid_rdy
:t20: rt_toggl_rcvd,toggl_is_lft -> toggl_is_rt,solnoid_toggld
:t21: rt_toggl_rcvd,toggl_is_rt,solnoid_in_use -> toggl_is_rt,\
                                                  solnoid_rdy
:t22: lft_toggl_rcvd,toggl_is_rt -> toggl_is_lft,solnoid_toggld
:t23: solnoid_NDRO_inpt,solnoid_rdy -> solnoid_rd_enabld, \
                                       solnoid_in_use
:t24: solnoid_rd_enabld,toggl_is_lft,solnoid_in_use -> \
                                          status_is_lft,\
                                         toggl_is_lft,solnoid_rdy
:t25: solnoid_rd_enabld,toggl_is_rt,solnoid_in_use -> \
                                          status_is_rt,\
                                          toggl_is_rt,solnoid_rdy
:t26: solnoid_toggld,solnoid_in_use -> \
                                    solnoid_toggld_outpt,solnoid_rdy
:t27: solnoid_disabl_inpt,solnoid_rdy -> solnoid_disabld,\
                                   solnoid_en_dis_outpt
:t28: solnoid_enabl_inpt,solnoid_rdy -> \
```

```
                                        solnoid_rdy,solnoid_en_dis_outpt
:t29: solnoid_enabl_inpt,solnoid_disabld -> solnoid_rdy,\
                                          solnoid_en_dis_outpt

/* following transitions, t30 - t34, consume incoming tokens while
*/
/* solenoid is in use. this prevents accumulation at the input
places */

:t30: solnoid_disabl_inpt,solnoid_in_use -> solnoid_in_use
:t31: solnoid_enabl_inpt,solnoid_in_use -> solnoid_in_use
:t32: toggl_rt_inpt,solnoid_in_use -> solnoid_in_use
:t33: toggl_lft_inpt,solnoid_in_use -> solnoid_in_use
:t34: solnoid_NDRO_inpt,solnoid_in_use -> solnoid_in_use

/* following transitions, t35 -t38, consume incoming tokens while
*/
/* solenoid is disabled */

:t35: solnoid_disabl_inpt,solnoid_disabld -> solnoid_disabld,\
                                           solnoid_en_dis_outpt
:t36: solnoid_NDRO_inpt,solnoid_disabld -> solnoid_disabld
:t37: toggl_lft_inpt,solnoid_disabld -> solnoid_disabld
:t38: toggl_rt_inpt,solnoid_disabld -> solnoid_disabld


:t39: solnoid_en_dis_outpt -> lkup_tbl_blt
:t40: lkup_tbl_blt -> ptr_cnt_initlzd
:t41: ptr_cnt_initlzd -> awaitng_ITL_status_chck
:t42: awaitng_ITL_status_chck -> ITL_NDRO_inpt
:t43: ITL_is_true -> ADC_enabl_inpt,ITL_lock_1
:t44: ITL_is_false -> ADC_enabl_inpt,no_ITL_lock_1

/* following transitions, t45 - t51, are contained in ADC module */

:t45: ADC_disabl_inpt,ADC_rdy -> ADC_disabl_rcvd
:t46: ADC_enabl_inpt,ADC_rdy -> ADC_enabl_rcvd
:t47: ADC_disabl_rcvd,ADC_disabld -> ADC_disabld,ADC_cmd_procssd
:t48: ADC_enabl_rcvd,ADC_disabld -> ADC_enabld,ADC_cmd_procssd
:t49: ADC_enabl_rcvd,ADC_enabld -> ADC_enabld,ADC_cmd_procssd
:t50: ADC_disabl_rcvd,ADC_enabld -> ADC_disabld,ADC_cmd_procssd
:t51: ADC_cmd_procssd -> ADC_outpt,ADC_rdy


:t52: ADC_outpt,ITL_lock_1 -> accel_inpttd
:t53: ADC_outpt,no_ITL_lock_1 -> accel_bias_inpttd
:t54: accel_inpttd -> ADC_disabl_inpt,ITL_lock_2
:t55: accel_bias_inpttd -> ADC_disabl_inpt,no_ITL_lock_2
:t56: ADC_outpt,no_ITL_lock_2 -> awaitng_ITL_status_chck
:t57: ADC_outpt,ITL_lock_2 -> awaitng_vel_updat
:t58: awaitng_vel_updat -> awaitng_sep_dist_updat
:t59: awaitng_sep_dist_updat,fourG_bst_occrrd -> sep_dist_updatd,\
                                                 fourG_bst_occrrd
:t60: sep_dist_updatd -> rchd_toggl_dist
:t61: sep_dist_updatd -> not_rchd_toggl_dist
```

```
:t62: rchd_toggl_dist -> ptr_cnt_incrmntd
:t63: not_rchd_toggl_dist -> awaitng_tmr_strt
:t64: ptr_cnt_incrmntd -> solnoid_NDRO_inpt,await_timr_delay
:t65: awaitng_tmr_strt -> tmr_running
:t66: tmr_running -> awaitng_elpsd_time_chck
:t67: awaitng_elpsd_time_chck -> tmr_running
:t68: awaitng_elpsd_time_chck -> tmr_wait_ovr
:t69: status_is_lft -> toggl_rt_inpt
:t70: status_is_rt -> toggl_lft_inpt
:t71: tmr_wait_ovr -> sep_dist_chckd
:t72: sep_dist_chckd -> rchd_toggl_dist
:t73: sep_dist_chckd -> ITL_is_true
:t74: solnoid_toggld_outpt -> ball_lck_toggld
:t75: ball_lck_toggld -> ball_rlsd
:t76: ball_rlsd -> one_third_incrs_of_intrptor
:t77: await_timr_delay -> rdy_for_wait200
:t78: rdy_for_wait200 -> awaitng_sep_dist_chck
:t79: awaitng_sep_dist_chck,iter_counter -> ITL_is_true
:t80: one_third_incrs_of_intrptor(3),SA_unlckd -> missl_armd
:t81: missl_armd -> missl_lckd_in_arm
:t82: missl_lckd_in_arm,snsr_detcts_tgt -> det_sig_rcvd
:t83: frnq_cap_chgd,det_sig_rcvd -> detonation

                /*   initial markings follow   */

<aircrft(1),SA_lckd(1),missl_on_rack(1),cmptr_off(1),no_ITL(1),ITL_
snsr_rdy(1),\
 toggl_is_rt(1),solnoid_rdy(1),ADC_rdy(1),ADC_disabld(1),\
 snsr_detcts_tgt(1),iter_counter(2) >
```

APPENDIX H

SUMMARY OF MODELING AND ANALYSIS METHODOLOGY

The recommended chronology and methodology for Petri net modeling and automated safety analysis of a software-controlled real-time system follows.

1. Study system functions. Have the designer explicitly identify all perceived hazardous conditions.

2. Study system and software flowcharts thoroughly. Attempt to reduce system scope to include only significant aspects pertinent to the stated hazards. If in doubt as to significance of any detail, include it.

3. Document system interfaces. Incorporate all flowcharts into a single Petri net system description. Abstract component internal functions by including only "black-box" component descriptions with external system interfaces. To increase readability, multiple component instantiations should be represented as separate "black-box" descriptions.

4. Once the initial Petri net system framework is complete, obtain verification from the designer. Study and model component functionality with Petri nets. As in system framework approach, incorporate only significant aspects. Attempt a second level of abstraction by further division of components into submodules and internal interfaces.

5. Following completion of individual component Petri net models, convert the nets to textual form. Any text editor can be used. Chapter VI gives detailed instructions for the conversion process.

6. Translate the component text file to internal Petri Net UTilities (P-NUT) format. Redirect output to a second file for later use. The appropriate translation command is **transl <file1> > <file1.pn>**. The .pn suffix identifies the file as an internal Petri net representation.

7. Build the reachability graph from the translated Petri net file using the Reachability Graph Builder (RGB) and redirect output. The proper command is **rgb [-bs] <file1.pn> > <file1.rg>**. The optional suffix b signals that the net is bounded at 127, while the s suffix signals that the net is safe, or bounded at 1. The .rg suffix denotes the file being in internal P-NUT reachability graph form. Note that the input file must be in internal P-NUT format.

8. The component's reachability graph can be printed in readable form using the Reachability Graph Printer (RGP). Redirect output to a new file. The proper command is **rgp <file1.rg> > <file1.g>**. The .g suffix is a recommendation only.

9. Study the reachability graphs and state spaces of each component. Verify that functionality has been accurately modeled.

10. If the component is complex and has a large reachability graph, use the Reachability Graph Analyzer to assist in the analysis. The command to invoke RGA is **rga <file1.rg> [function libraries]**. Notice that user-defined function libraries may be invoked and used with RGA. The input file to RGA must be in internal P-NUT reachability graph format. Chapter VI gives several detailed examples of analysis expression syntax for the RGA language.

11. After validating component models, incorporate them into a textual file version of the overall net. Repeat steps 6 through 10 for the system model. The final reachability graph may contain several thousand states necessitating analysis solely with the RGA. Translation of safety analysis questions to RGA net terminology and syntax is discussed in Chapter VI.

LIST OF REFERENCES

Department of Information and Computer Science, University of California, Irvine, CA, Report 85-06, *Computer-Aided Analysis of Concurrent Systems*, by E. T. Morgan and R. Razour, 8 Feb. 1985.

Department of Information and Computer Science, University of California, Irvine, CA, Report 87-04, *RGA User's Manual Version 2.3*, by E. T. Morgan, 13 Jan. 1987.

Department of Information and Computer Science, University of California, Irvine, CA, Report 86-25, *A Guided Tour of P-NUT (Release 2.2)*, by R. R. Razour, Jan. 1987.

Ericson, C. A., "Software and System Safety", *Proceedings of the 5th International System Safety Conference* (Denver, CO), vol. 1, part 1, System Safety Society, Newport Beach, CA, pp. III-B-1 to III-B-1, 1981.

Hayward, D. F., "A Practical Application of Petri Nets in the Software Safety Analysis of a Real-Time Military System", M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1987.

Jahanian, F. and Mok, A. K., "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Transactions on Software Engineering*, SE-12, 9 Sept. 1986, pp. 890-904.

Konakovsky, R., *Safety Evaluation of Computer Hardware and Software*. Proceedings of Compsac '78, IEEE, New York, pp. 559-564, 1978.

Lauber, R., "Strategies for the Design and Validation of Safety-Related Computer-Controlled Systems", *Real-Time Data Handling and Process Control*, G. Meyer, ed., North-Holland Publishing, Amsterdam, pp. 305-310, 1980.

Leveson, N. G., "Software Safety: Why, What, and How", *Computing Surveys*, vol. 18, no. 2, June 1986.

Leveson, N. G., and Stolzy, J. L., "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, Mar. 1987.

McVay, J., *Point Paper on Conducting a Design, Development, and Safety Review of a Guided Missile Safety-Arming Device Utilizing a Noninterrupted Explosive Train*, NWC TM (draft), NWC, China Lake, CA, 1987.

MIL-STD-1316C, *Safety Criteria for Fuze Design*, Dept. of Defense, GPO, Wash., DC, 3 Jan. 1984.

MIL-STD-1574A (USAF), *System Safety Program for Space and Missile Systems*, Dept. of Air Force, GPO, Wash., DC, 15 Aug. 1979.

MIL-STD-882B Notice 1, *System Safety Program Requirements*, Dept. of Defense, GPO, Wash., DC, 1 July 1987.

MIL-STD-SNS (Navy), *Software Nuclear Safety* (draft), available from Naval Weapons Evaluation Facility, Kirtland Air Force Base, NM, 1986.

Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Petri, C., *Kommunikation mit Automaten*, Ph.D. dissertation, University of Bonn, Bonn, West Germany, 1962.

Roland, H. E., and Moriarity, B., *System Safety Engineering and Management*, Wiley, NY, 1983.

Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F., *Fault Tree Handbook*, US Nuclear Regulatory Commission, Report NURTEG-0492, Jan. 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Commander (Code 34) Naval Air Test Center Patuxent River, MD 20670	2
4.	Commander (Code 31C) Naval Weapons Center China Lake, CA 93555	2
5.	Commander (Code 3353) Naval Weapons Center China Lake, CA 93555	5
6.	LCDR John Yurchak, USN (Code 52YY) Naval Postgraduate School Monterey, CA 93943-5002	3
7.	Daniel Davis MBARI 160 Central Avenue Pacific Grove, CA 93950	1
8.	Nancy Leveson Department of Information and Computer Science University of California Irvine, CA 92717	1
9.	Rami Razouk Department of Information and Computer Science University of California Irvine, CA 92717	1

10. Duston Hayward 1
Naval Ocean Systems Command
Code 423
271 Catalina Boulevard
San Diego, CA 92152-5000

11. Robert Wasilausky 1
Naval Ocean Systems Command
Code 423
271 Catalina Boulevard
San Diego, CA 92152-5000

12. Uno Kodres (Code 52) 1
Naval Postgraduate School
Monterey, CA 93943-5002

13. LT Alan D. Lewis, USN 3
c/o LTG B. L. Lewis, USA (Ret.)
1928 Relda Court
Falls Church, VA 22043